Tampereen yliopisto

Osku Haavisto

MATKAPUHELINVERKKOJEN ANALYSOINTIOHJELMISTO

Informaatioteknologian ja viestinnän tiedekunta
Diplomityö
Marraskuu 2019

TIIVISTELMÄ

Osku Haavisto: Matkapuhelinverkkojen analysointiohjelmisto
Diplomityö
Tampereen yliopisto
Tietotekniikan diplomi-insinöörin tutkinto-ohjelma
Marraskuu 2019

Matkapuhelinverkkojen tukiasemista sekä niiden hallinnoimista puheluista ja tiedonsiirroista kerätään jatkuvasti huomattavat määrät dataa. Erityisesti erilaisia virhetapahtumia koskevat tiedot kiinnostavat paljon sekä operaattoreita että tukiasemien laitevalmistajia. Jos tietyllä alueella esimerkiksi katkeaa paljon puheluita, voidaan virhetiedoista selvittää syy siihen ja korjata se lisähäiriöiden välttämiseksi. Tietojen tulkitseminen ei kuitenkaan ole yksinkertaista. Data on usein matalatasoista, josta yksinään ei välttämättä pystytä päättelemään paljoakaan. Tätä tietoa on myös erittäin paljon, joten sen käsittelemiseen vaaditaan tehokkaita ohjelmistoja.

Tässä diplomityössä tutustutaan matkapuhelinverkkojen analysointiohjelmistoon, joka pyrkii löytämään ja tunnistamaan virhetilanteita verkoista reaaliajassa. Työ keskittyy kirjoittajan kehittämään ohjelmiston komponenttiin, joka on vastuussa tiedon muokkaamisesta ja analysoinnista sekä ohjelmiston sisäisen tiedon välittämisestä. Tämä ohjelma myös vastaa matkapuhelinverkosta löydettävistä poikkeamista tai virheistä johtuvien hälytysten hallinnoinnista. Ohjelmistoon kuuluu myös GUI-ohjelma ja raskaan laskennan suorittava laskentaohjelma. Työssä käydään läpi ohjelman suunnittelua, toteutusta ja testausta sekä arvioidaan, kuinka hyvin näissä onnistuttiin.

Ohjelmistolle annetut vaatimukset saavutettiin. Ohjelmiston reaaliaikainen verkon seuranta saatiin aikaiseksi hyödyntämällä tiedon käsittelyssä striimeja tietokantojen sijaan. Diplomityössä toteutettu ohjelma tarjosi REST API:n tulosten hakemiseen ja ohjelmien ohjaukseen. Vaikka REST API ei täyttänyt kaikkia REST arkkitehtuurin vaatimuksia, toimi se hyvin tässä käyttötarkoituksessa. Ohjelman kyky aloittaa automaattisesti tarkka keräys tukiasemissa vikatilanteissa on saanut kiitosta. Yleisestikin ohjelmaan ja ohjelmistoon ollaan tyytyväisiä.

Avainsanat: Matkapuhelinverkko, tukiasema, REST, ohjelmistokehitys

ABSTRACT

Osku Haavisto: Cellular Network Analysis Software
Master of Science Thesis
Tampere University
Master's Degree Programme in Information Technology
November 2019

Significant amounts of data are continuously collected from base stations of cellular networks and from calls and data communications managed by them. Information on various error events is of particular interest to both operators and base station equipment manufacturers. For example, if many calls are lost in a certain area, the error information can be used to determine the cause and correct it to avoid further interference. However, interpreting the information is not straightforward. Data is often low-level, which alone may not be able to infer much. This information is also very large and requires efficient software to handle it.

This thesis introduces mobile network analysis software, which seeks to find and identify error situations in networks in real time. The work focuses on the software component developed by the author, which is responsible for editing and analyzing data and for communicating information internally within the software. This program is also responsible for managing alarms due to anomalies or errors found on the cellular network. The software also includes a GUI program and a heavy computing program. The work goes through the design, implementation and testing of the program and evaluates how well these have been achieved.

Requirements for the software were met. Real-time network monitoring of the software was achieved by utilizing streams instead of databases to process data. The program developed in this thesis provided a REST API for results retrieval and program control. Although the REST API did not meet all the requirements of the REST architecture, it worked well for this purpose. The ability of the program to automatically start accurate collection at base stations in case of failure has been praised. Overall, people are satisfied in the program and software.

Keywords: cellular network, mobile network, base station, REST, software development

ALKUSANAT

Tämä diplomityö tehtiin Nokia Oyj:lle. Haluan kiittää tiimiäni mahdollisuudesta tehdä tämä diplomityö sekä mahtavasta ja mielekkäästä työilmapiiristä.

Kiitän kaikkia perheenjäseniäni, ystäviäni ja työkavereitani minulle antamastanne tuesta ja avusta viimeisen vuoden aikana. Erityisesti haluan kiittää työn ohjaajana ja tarkistajana toiminutta professori Kari Systää kaikista neuvoista ja kommenteista.

Tampereella,14.11.2019

Osku Haavisto

SISÄLLYSLUETTELO

KUVALUETTELO

LYHENTEET JA MERKINNÄT

.

API	ohjelmointirajapinta (Application Programming Interface)
BNF	notaatiotekniikka (Backus−Naur form/Backus normal form)
BTS	tukiasema (Base Tranceiver Station)
DevOps	ohjelmistokehityksen (software development, Dev) ja IT-palveluiden information-technology operations, Ops) yhdistelmä
GUI	graafinen käyttöliittymä (Graphical User Interface)
HealthCheck	työssä kuvatun matkapuhelinverkon analysointiohjelmiston nimi
HcEngine	työssä käsitelty laskentaohjelma (HealthCheck Engine)
HcGUI	työssä käsitelty graafinen käyttöliittymäohjelma (HealthCheck GUI)
HcOSA	työssä toteutettu analysointiohjelma (HealthCheck Operating Servicing Analyzer)
HTTP	tiedonsiirtoprotokolla (Hypertext Transfer Protocol)
HTTPS	suojattu tiedonsiirtoprotokolla (Hypertext Transfer Protocol Secure)
JSON	tiedostomuoto tiedonvälitykseen (JavaScript Object Notation)
REST	arkkitehtuurimalli (REpresentational State Transfer)
TCP	tietoliikenneprotokolla (Transmission Control Protocol)
UDP	tietoliikenneprotokolla (User Datagram Protocol)

1. JOHDANTO

Matkapuhelimet ovat kasvaneet yhdeksi välttämättömimmiksi perustarpeiksi 2000-luvun puolella. Ihmisten elämät ovat yhä enemmissä määrin kytkeytyneet puhelimiin ja niiden tarjoamiin palveluihin. Tutkimuksen mukaan [2] vuoden 2018 lopulla 5,1 miljardilla ihmisellä oli ainakin yksi matkapuhelinsopimus ja yhteensä näitä matkapuhelinsopimuksia hyödyntäviä laitteita oli 7,9 miljardia. Näiden laitteiden toiminta ja käyttö pohjaavat pitkälti kommunikointiin toisten laitteiden välillä ja täten ne ovat riippuvaisia matkapuhelinverkoista. Mikäli verkko kaatuisi, ei matkapuhelimella pystyisi soittamaan tai selaaman internettiä eikä se vastaanottaisi yhteyksiä muualtakaan. Matkapuhelinverkon kaatuminen on erityisen harvinaista, mutta yleisempää on, että tietyllä alueella puhelut katkeavat taikka tiedonsiirtonopeudet ovat erittäin alhaisia. Syinä tällaisiin tapahtumiin ovat pääasiassa operaattorien tukiasemien puutteellinen kattavuus tai toiminta. Verkon toiminnan vakaudella ja laadulla on laitteiden kautta suuri merkitys ihmisen arkielämässä. Täten onkin tärkeää varmistaa, että ongelmia ei ilmene verkoissa tai ihmiset tulevat erittäin nopeasti tyytymättömiksi.

Verkkojen tilasta ja toiminnasta saadaan tietoa erilaisten parametrien välityksellä. Jos verkossa on ongelmia, saadaan vian syy selville usein näistä arvoista. Yhtä tukiasemaa kohden näitä parametrejä on muutama tuhat ja isommissa verkoissa voi olla jopa kymmeniä tuhansia tukiasemia. Näin ollen verkon parametrimäärä voi nousta useisiin miljooniin. Tällaista tietomäärää on vaikea ymmärtää ja hallita. Avuksi tarvitaan erityisesti tähän tarkoitukseen suunniteltuja ja toteutettuja ohjelmistoja.

Vikojen selvittämiseen tarkoitetut ohjelmistot ovat perinteisesti olleet hitaita. Vikojen löytämiseen on usein kulunut vuorokausi. Olemassa olevat ratkaisut sisällyttävät myös paljon manuaalista työtä. Parametridatan saamiseksi työntekijän täytyy usein matkustaa paikanpäälle ja analysoida tieto siellä. Tämä on sekä kallista että hidasta. Mitä nopeammin matkapuhelinverkon hallitsija saa tiedon siinä olevista puutteista, sitä nopeammin ne voidaan paikata.

Tässä diplomityössä tutustutaan matkapuhelinverkkojen analysointiohjelmistoon, joka pyrkii löytämään ja tunnistamaan virhetilanteita verkoista reaaliajassa. Työ keskittyy kirjoittajan kehittämään ohjelmiston komponenttiin, joka on vastuussa tiedon muokkaamisesta ja analysoinnista sekä ohjelmiston sisäisen tiedon välittämisestä. Työssä käydään

läpi ohjelman suunnittelua, toteutusta ja testausta sekä arvioidaan, kuinka hyvin näissä onnistuttiin.

Luvussa 2 käydään läpi matkapuhelinverkon rakennetta yleisesti ja kerrotaan lyhyesti tukiasemista ja niiden parametreistä. Luvussa 3 avataan lähtökohtia ohjelmiston tekemiseen ja käydään läpi sille osoitettuja vaatimuksia. Luku 4 keskittyy ohjelman suunnittelun läpikäyntiin ja luvussa 5 taas kerrotaan toteutuksesta. Luku 6 kertoo ohjelman ja ohjelmiston testauksesta. Luvussa 7 pohditaan ohjelman onnistumista ja kerrotaan jatkototeutuksesta. Luku 8 on yhteenveto.

2. TAUSTA

Ensiksi kerrottaan hieman radioverkoista yleisellä tasolla. Seuraavaksi käydään läpi radioverkkojen toimivuuden seuraamista erilaisten parametrien avulla, joita on paljon. Viimeiseksi esitetään ongelma siitä, kuinka löydetään parametrijoukkoja, jotka osoittavat selkeästi radioverkon kunnon.

2.1 Matkapuhelinverkon rakenne

Matkapuhelinverkko koostuu neljästä pääkomponentista, joiden yhteistoiminta mahdollistaa asiakkaiden palvelun:

1. Yleinen puhelinverkko (Public switched telephone network, PSTN)
2. Matkapuhelinkeskus (Mobile switching center, MSC)
3. Tukiasema
4. Päätelaite

Yleisen puhelinverkon tehtävä on yhdistää erinäisten vaihto- ja kaukoverkkojen avulla kaikki paikalliset matkapuhelinverkot yhteen ja täten mahdollistaa kommunikointi ympäri maailmaa. Matkapuhelinkeskus on vastuussa puheluiden siirtämisestä yleiseen puhelinverkkoon. Keskuksen muita tehtäviä ovat puheluiden hallinta, tietojen keräys ja päätelaitteiden paikannus. Tätä varten sillä on käytössään kaksi eri rekisteriä: kotirekisteri (Home location register, HLR) ja vierasrekisteri (Visitor location register, VLR). Kotirekisteri on päätietokanta, joka sisältää kattavat tiedot operaattorin kaikista tilaajista. Näitä tietoja ovat esimerkiksi laitteen tunnistenumero, puhelinnumero ja viimeisin sijainti. Tämä rekisteri voi olla kuitenkin erityisen iso ja sen jatkuva-aikainen käyttäminen keskuksessa olisi erittäin raskasta. Vierasrekisterissä on sen sijaan ainoastaan puhelun muodostamiseen tarvittavat tiedot matkapuhelinkeskuksen alueella olevista päätelaitteista. Jotta nämä tiedot olisivat ajan tasalla ja paikkaansa pitäviä, päätelaitteet lähettävät tarvittavia tietoja aina tietyn ajoin omatoimisesti ja vierasrekisteri noutaa näitä vastaavat tiedot kotirekisteristä. Nämä tiedot ovat elintärkeitä, sillä ilman niitä ei tiedettäisi minkä tukiaseman alueella puhelin sijaitsee eikä puhelin voisi vastaanottaa puheluita. [3]

Tukiasemat ovat ensimmäinen osa verkkoa, joihin päätelaitteet, kuten matkapuhelimet ja tabletit, ovat yhteydessä. Näiden välinen liikenne tapahtuu korkeataajuisten radioaaltojen, mikroaaltojen, avulla. Muuten matkapuhelinverkoissa liikenne tapahtuu pääasiassa maan alla olevien valokuitukaapelien välityksellä. Aluetta, jolta päätelaitteet saavat

yhteyden samaan tukiasemaan, kutsutaan soluksi. Tukiaseman muodostaman solun kokoon ja muotoon vaikuttavat tukiasemassa olevan lähettimen teho ja taajuus sekä antennin suunta ja muoto. Kokoon vaikuttaa myös erityisesti maasto ja tukiaseman sijainti. Tiheän asutuksen seudulla palveltavia päätelaitteita on paljon. Koska tukiasema voi palvella vain tiettyä määrää asiakkaita kerrallaan, ei solu voi olla kovin laaja. Mikäli tukiasemaan yhdistyisi useampi päätelaite kuin se pystyy tukemaan, jouduttaisiin mahdollisesti katkaisemaan vielä olemassa olevia puheluita tai datayhteyksiä. Kaupungeissa onkin täten monia tukiasemia ja solujen koot ovat pieniä. Solut ovat myös osittain päällekkäin tarjoten mahdollisuuden päätelaitteelle valita parhain tukiasema palvelemaan sitä kyseisestä sijainnista. Harvaan asutulla seudulla palveltavia asiakkaita on taas vähän ja tukiasemien vähentämiseksi niiden tehoja on nostettu isojen solujen luomiseksi ja kattavuuden takaamiseksi. [3,13]

2.2 Vierekkäiset solut ja yhteysvastuun siirto

Fysiikan lakien mukaisesti tukiaseman lähettämät radioaallot vaimenevat etäisyyden kasvaessa, toisin sanoen kuuluvuus on parempi lähellä tukiasemaa. Vaikka tukiaseman radioaallot ovat heikentyneet huomattavasti edettyään naapurisoluihin, käyttävät vierekkäiset solut eri taajuuksia, jotteivat ne häiritsisi toisiaan. Tämä syy aiheuttaa kuitenkin sen, että jos päätelaite siirtyy tarpeeksi kauas tukiasemastaan toisen solun alueelle puhelun ollessa käynnissä, puhelu tulee katkeamaan signaalin heikennettyä tarpeeksi. Tämän estämiseksi on kehitetty toiminto nimeltään yhteysvastuun vaihto, josta käytetään usein sen englannin kielen termiä handoff tai handover (HO). Tässä tekniikassa radiotaajuutta vaihdetaan puhelun aikana vastaamaan viereisen solun taajuutta keskeyttämättä puhelua. Taajuuden ollessa oikea voidaan sekä puhelu että vastuu laitteen hallinnoimisesta siirtää toisen solun tukiasemalle. [3]

Yhteysvastuun vaihto ei kuitenkaan aina suju täydellisesti. Ongelmia voi tulla signaalin voimakkuuden mittaamisessa, siirron aloittamisajankohdan päätöksenteossa tai varsinaisessa siirrossa. Signaalin voidaan mitata olevan voimakkaampi kuin se todellisuudessa on ja vaihdosta ei tehdä ennen kuin puhelu on jo keskeytynyt. Arvo, joka kertoo, milloin tukiaseman tulisi aloittaa tekemään yhteysvastuun vaihdosta, voi olla väärin, tai tilanne on normaalista poikkeava. Esimerkiksi arvo voisi kertoa, että siirros tulee aloittaa, kun signaalin voimakkuus on 20%. Jos päätelaite on autossa ja liikkuu nopeammin kuin oletetaan, puhelin joutuu liian kauas tukiasemasta ennen kuin handover on valmis. [3]

2.3 Tietoliikennearkkitehtuuri

Tietoliikennearkkitehtuurissa on kolme oleellista osaa: käyttäjä- tai datataso (user plane, data plane), ohjaustaso (control plane) ja hallintotaso (management plane). Näillä kaikilla on omat tehtävänsä ja ne kuljettavat eri tyyppistä tietoa. Datatasolla kulkee verkossa olevien käyttäjien luoma liikenne. Ohjaustasolla kulkee reitittimien välinen ohjausliikenne. Hallintotasolla kulkee hallintaan ja konfigurointiin liittyvä liikenne. [4]

Matkapuhelinverkossa datatasolla liikkuu varsinainen puhelun tieto. Ohjaustasolla kerrotaan puhelun muodostamiseen ja reititykseen tarvittavat tiedot Ohjaustason tiedoista nähdään puhelun aloittamisaika ja lopetustietojen lisäksi muun muassa sen reitittyminen eri tukiasemien välillä. Ohjaustason tietojen perusteella voidaan tehdä paljon analyysia verkon tilasta. [4]

3. LÄHTÖKOHTA

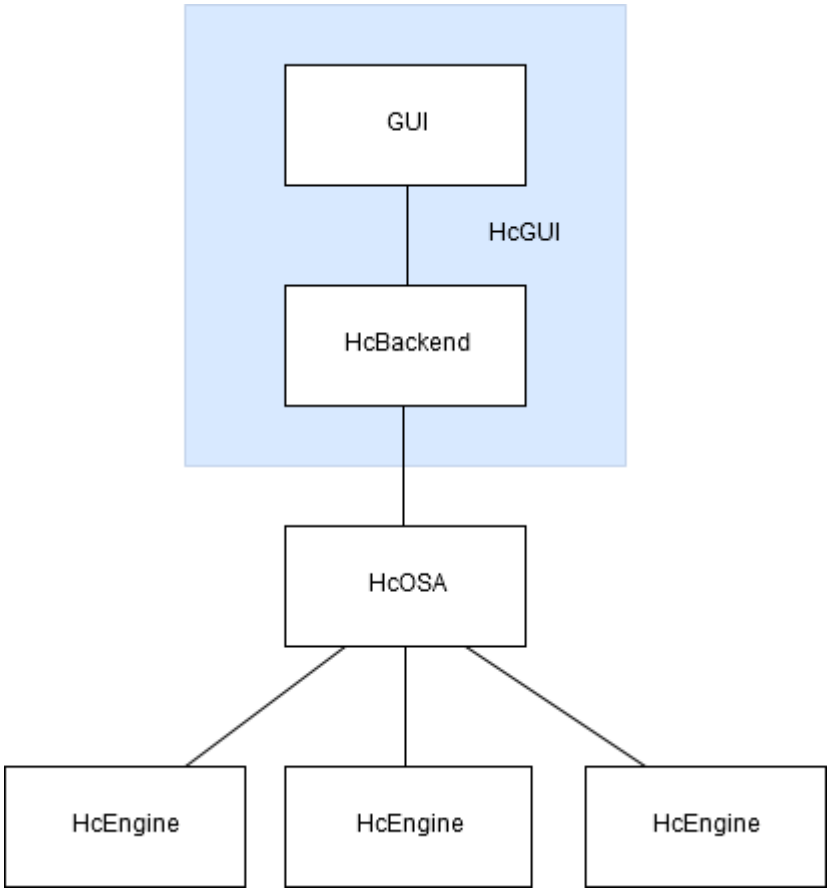
Matkapuhelinverkon toimintatilan näkeminen ja ymmärtäminen on erittäin tärkeää verkon omistaville teleoperaattoreille. Operaattoreille verkkojen vakaa ja laadukas toiminnallisuus sekä niiden kattavuus ovat oleellisimmat kilpailuominaisuudet. Mikäli puhelut alkaisivat katkeamaan taikka kuuluvuus olisi erittäin heikkoa joillakin alueilla, laskisi asiakastyytyvyys ja teleoperaattorin bisnes kärsisi. Tämän vuoksi virheiden ja poikkeamien ajoissa havaitseminen verkossa on elintärkeää operaattoreille. Poikkeamien löytämiseen tarkoitetut erinäiset ja eritasoiset analysointipalvelut mahdollistavat operaattorin tekemään tarvittavat toimenpiteet haittavaikutusten minimoimiseksi.

Nykyiset vianetsintäohjelmat ovat hitaita ja raskaita. Tieto kerätään ja tallennetaan suuriin tietokantoihin, josta jälkikäteen voidaan hakea arvoja tutkittavaksi. Usein näihin tietokantoihin ei ole pääsyä muualta kuin suoraan asiakkaan palvelinkeskuksista ja useasti virhetilanteiden ymmärtämisessä tarvitaan asiantuntija-apua. Tällöin ainoa vaihtoehto on matkustaa paikan päälle, mikä on sekä kallista että hidasta.

Työssä esiteltävän ohjelmiston on tarkoitus pystyä laskemaan ja analysoimaan verkosta tulevaa dataa reaaliajassa sekä tarjoamaan graafinen käyttöliittymä tulosten tutkimiseen mistä tahansa päin maailmaa. Ohjelmisto on perusidealtaan healthcheck eli se pyrkii kertomaan, toimiiko tutkittava kohde odotetusti. Healthcheck:lle tyypillisen yksittäisen tai säännöllisen tarkastuksen suorittamisen sijaan ohjelmisto tarkkailee matkapuhelinverkkoa jatkuva-aikaisesti. Ohjelmisto koostuu useammista pienemmistä ohjelmista, joilla on omat tehtävänsä. Näiden eri ohjelmien ohjaamiseksi tarvitaan oma ohjelma, joka on tämän diplomityön aihe. Ohjelman tehtävä on muiden ohjelmien ohjaamisen lisäksi kerätä ja yhdistää tietoa ohjelmilta, tehdä omaa analysointia tälle tiedolle ja lähettää sitä eteenpäin.

Alustava idea ohjelmiston koostumuksesta jakautui kolmeen selkeään omaan osaansa. Ensimmäinen ohjelma (HcEngine, Healthcheck Engine tai lyhyesti Engine) hoitaisi laskennan ja alustavan analyysin. Näitä laskennan suorittavia ohjelmia voisi olla useampia ja toisen ohjelman (HcOSA, Healthcheck Operating Servising Analyzer) tehtävänä on yhdistää näiltä tuleva tieto, tehdä analyysi ja lähettää se kolmannelle ohjelmalle (HcGUI, Healthcheck GUI tai lyhyesti GUI). Tämän avulla käyttäjä voisi ohjata ohjelmiston toimintaa ja nähdä tulokset. HcGUI muodostui frontendistä (GUI) ja backendistä (HcBackend). Tässä työssä toteutetaan HcOSA-ohjelma. Koko ohjelmistoa kutsuttiin nimellä

HealthCheck ja sen yleisarkkitehtuuri on mallinnettu kuvassa 1. HcEngine-ohjelmien lukumäärä on kuvassa kolme, mutta se voi vaihdella väliltä 1-n.

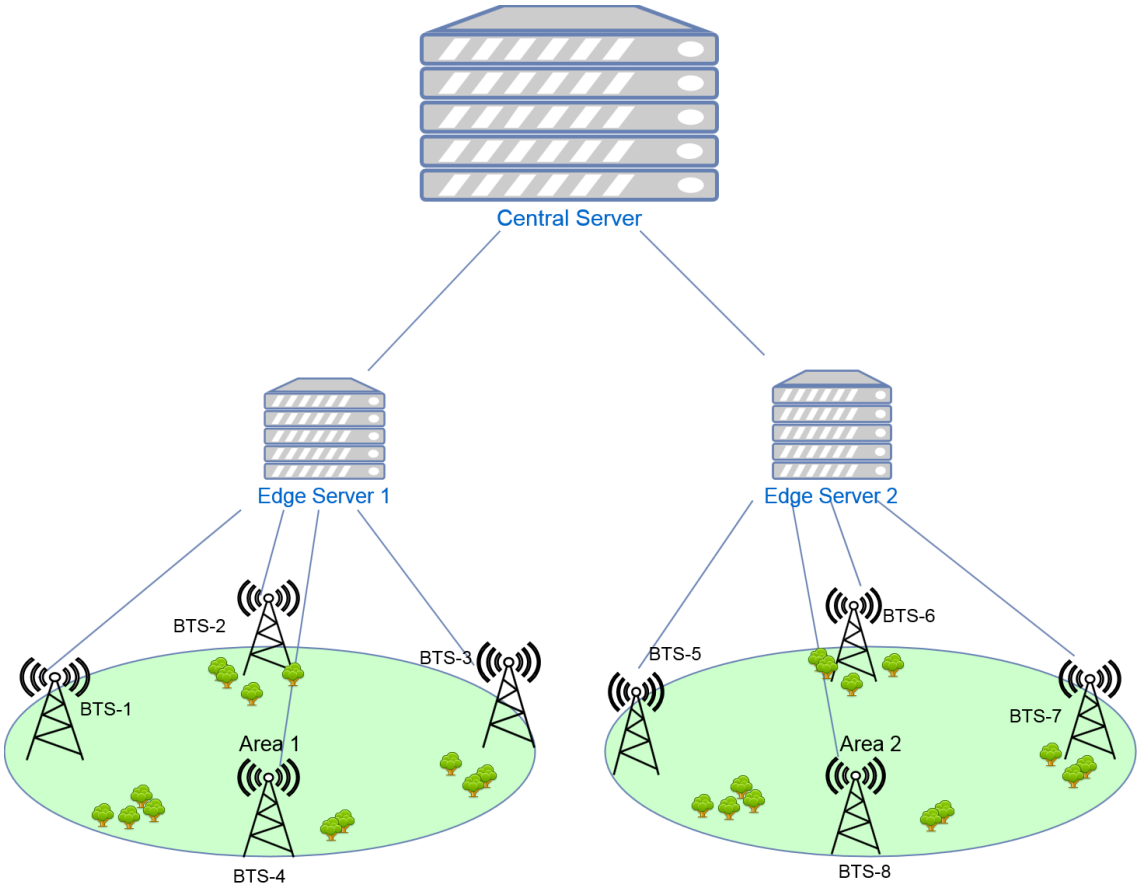


Kuva 1. *HealthCheck-ohjelmiston yleisarkkitehtuuri.*

3.1 Ympäristö

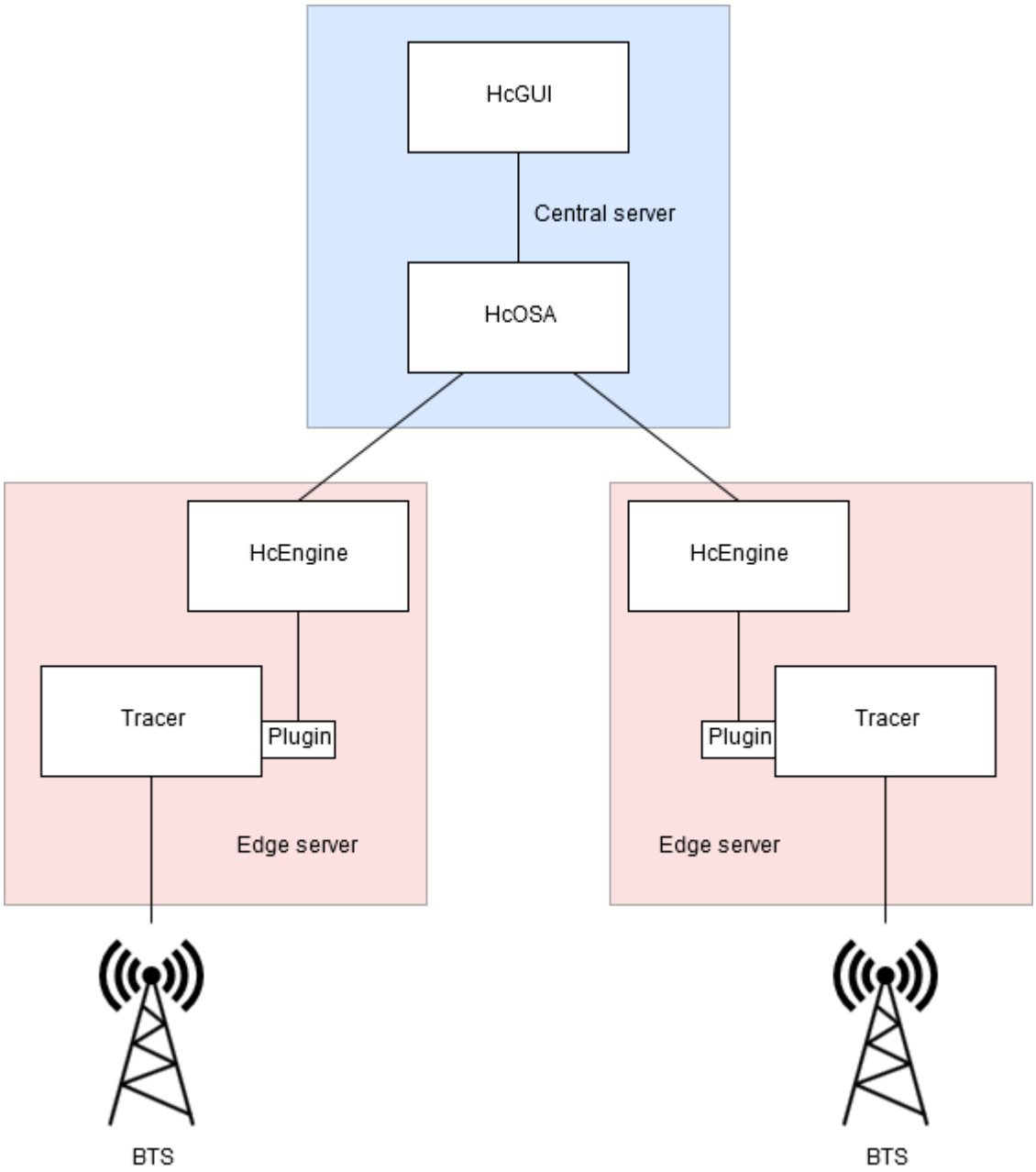
Matkapuhelinverkoissa on yleensä yksi päädatakeskus, jossa erilaiset ohjelmistot käsittelevät dataa. Tukiasemasta sekä sen hallinnoimista puheluista ja tiedonsiirroista kerätään jatkuvasti huomattava määrä dataa. Koska verkossa voi olla jopa kymmeniä tuhansia tukiasemia, on muodostuvan tiedon määrä niin suuri, että sen siirtäminen kauas päädatakeskukseen olisi hidasta ja kallista. Tämän sijasta tukiasemat on ohjattu lähettämään dataa tietyille pienempiin ja lähempänä oleviin datakeskuksiin. Siellä niin kutsutuilla reunaservereillä (edge server) datan vastaanottaa ohjelma, joka suodattaa siitä halutun tiedon ja lähettää sen eteenpäin jatkokäsittelyyn päädatakeskukseen keskusser-

veri (central server). Verkossa on siis monia reunaservereitä, mutta ainoastaan yksi keskusserveri. Kuvassa 2 näkyy kaksi aluetta, joilta tukiasemat (BTS, base transceiver station) lähettävät tietoa niistä vastaavalle reunaserverille, joka taas lähettää tiedon keskusserverille.



Kuva 2. *Tukiasemien suhde reunaservereihin ja keskusserveriin.*

Reunaserverillä sijaitseva Tracer-ohjelma vastaanottaa ja purkaa tukiasemilta tulevan pakatun tiedon. Tämän jälkeen ohjelmassa kiinniolevat liitännäisohjelmat (plugin) suodattavat ja muokkaavat tietoa eri käyttötarkoituksiin sopiviin muotoihin. Muokatun tiedon ne lähettävät sitä pyytäville ohjelmille. HcEngine-ohjelma saa tietonsa yhdeltä tämänkaltaiselta liitännäisohjelmalta. Laskentaohjelma sijaitsee myös samalla reunaserverillä datansiirron minimoimiseksi. HcEngine tarvitsee lähettää ainoastaan tulokset keskusserverillä sijaitsevalle HcOSA:lle. Tämä yhdistää kaikilta reunaservereiltä tulevien laskentaohjelmien tulokset ja tarjoaa ne myös keskusserverillä sijaitsevalle HcGUI:lle. Kuva 3 esittää HealthCheck-ohjelmiston eri osien jakautumisen reuna- ja keskusserverien välille. Siinä kuvataan myös datan saapuminen HcEngine:lle.



Kuva 3. *HealthCheck-ohjelmiston jakautuminen reuna- ja keskusserverille ja datan saaminen tukiasemilta*

3.2 Vaatimukset

Tuotetta lähdettiin kehittämään sisäisesti eli vaatimukset eivät tulleet suoraan asiakkaalta. Lähtökohtana oli toki ymmärrys edellisistä vastaavista tuotteista ja asiakkaiden niihin kohdistamista vaatimuksista ja toiveista, mikä muodosti hyvän perustuksen tuotteen suunnittelulle. Vaatimukset muodostuvat toimintaympäristöstä, ohjelmiston varsi-

naisen toiminnallisuuden vaatimuksista ja vaatimuksista koskien yhteensopivuutta muihin tuotteisiin. Projektin edetessä uusia vaatimuksia ilmeni ja vanhat muuttuivat, kuten on tavallista ohjelmistoprojektissa.

3.2.1 Toimintaympäristön vaatimukset

Ohjelmiston päävaatimuksena oli sen kyky toimia Linux-ympäristössä. Muut tuotteet olivat myös siirtymässä tai jo siirtyneet Linux-ympäristöön sen tarjoaman paremman suorituskyvyn sekä reaaliaikaisuuden takia ja oli tärkeää, että myös analysointiohjelmisto olisi tarjolla uudessa toimintaympäristössä. Siirtymisessä Linux-ympäristöön oli myös syynä sen tarjoama parempi tuki Docker-konteille. Vaatimus ohjelmiston toimittamiseen konteissa pohjasi ohjelmiston samanlaiseen toimintaan eri ympäristöissä ja asentamisen sekä käyttöönoton helpottumiseen. Tämä on erittäin tärkeää DevOps-malliseen toimintaan, mitä tässä projektissa käytettiin. Aluksi ei ollut mitään erityisvaatimuksia, minkälaista peruspohjaa (base image*)* Docker-konteissa käytti. Toteutuksen oltua meneillään jo hyvän aikaa, tuli vaatimus käyttää Red Hat:in toimittamaa peruspohjaa. Vaatimus johtui yhtiöiden välillä olevasta sopimuksesta ja täten Red Hat tarjosi tukea omille tuotteilleen. Peruspohjan vaihdos johti vanhempien kääntäjien käyttöönottoon, mikä taas johti pieniin koodin muokkauksiin uusimpien rakenteiden tukemisen poistumisen myötä.

3.2.2 Ohjelmiston toiminnallisuuden vaatimukset

Ohjelmiston lähtökohtana oli tarjota kyvykkyys seurata ja analysoida matkapuhelinverkkojen terveydentilaa reaaliajassa. Reaaliajalla ja reaaliaikaisuudella tarkoitetaan tässä työssä ohjelmiston tuottamien tulosten mahdollisimman hyvää vastaavuutta nykyhetkeen eli verkossa havaittu virhe ilmoitetaan käyttäjälle mahdollisimman nopeasti. Vaatimukset graafisesta esityksestä ja ajantasaisesta laskennasta johtivat alussa ohjelmistoon jakamiseen kolmeen selkeään osaa: laskentaosaan (HcEngine), kokoaja- ja analysointiosaan (HcOSA) sekä GUI-osaan (HcGUI). Tarkemmin vaatimukset HcOSA:lle olivat alun perin toteuttaa rajapinta GUI-elementille ja kyky kommunikoida yhden tai useamman laskentayksikön kanssa. GUI-rajapinnan kautta HcOSA:lle tuli pystyä antamaan tehtäviä, kuten esimerkiksi kaavojen luominen tai laskutuloksien hakeminen. Laskentayksiköiden kanssa kommunikointi tuli mahdollistaa erilaisten laskutehtävien antamisen ja vastauksien saamisen HcEngine:iltä. Näitä vastauksia tuli osata yhdistää, mikäli niitä tuli useammasta yksiköstä ja muuntaa vastaus GUI:lle sopivaan muotoon. HcOSA:n tuli tarjota myös rajapinta mahdollisille toisille ohjelmille, jotka haluaisivat hyödyntää ohjelmiston tietoa. Myöhemmin tuli lisävaatimuksia, joiden mukaan HcOSA:n tuli kyetä käs-

kyttämään toista ohjelmaa tarkempaan tiedonkeräykseen, mikäli jokin raja-arvo on ylitetty. Samoin tuli pystyä välittämään erilaisia hälytyksiä, niistä vastaavalle päätuotteelle. Näiden lisäksi oli myös paljon pienempiä vaatimuksia ja muutoksia, jotka vaikuttivat esimerkiksi GUI:lle tarkoitettuun rajapintaan sekä HcOSA:n ja HcEngine:n väliseen kommunikointiin.

3.2.3 Yhteensopivuusvaatimukset

Usean ohjelman kanssa kommunikointi luo luonnollisesti yhteensopivuusvaatimuksia näihin ohjelmiin. Kaikkia kolmea ohjelmaa (HcGUI, HcOSA, HcEngine) suunniteltiin ja rakennetiin samaan aikaan. Tämä sekä helpotti että vaikeutti rajapintojen suunnittelua ja toteutusta. Helpottavana tekijä oli rajapintojen muokattavuus kaikkien osapuolien kannalta sopivaksi, mutta taas varjopuolena rajapinnat elivät paljon ja ei ollut varmuutta rajapintojen valmiudesta.

4. SUUNNITTELU

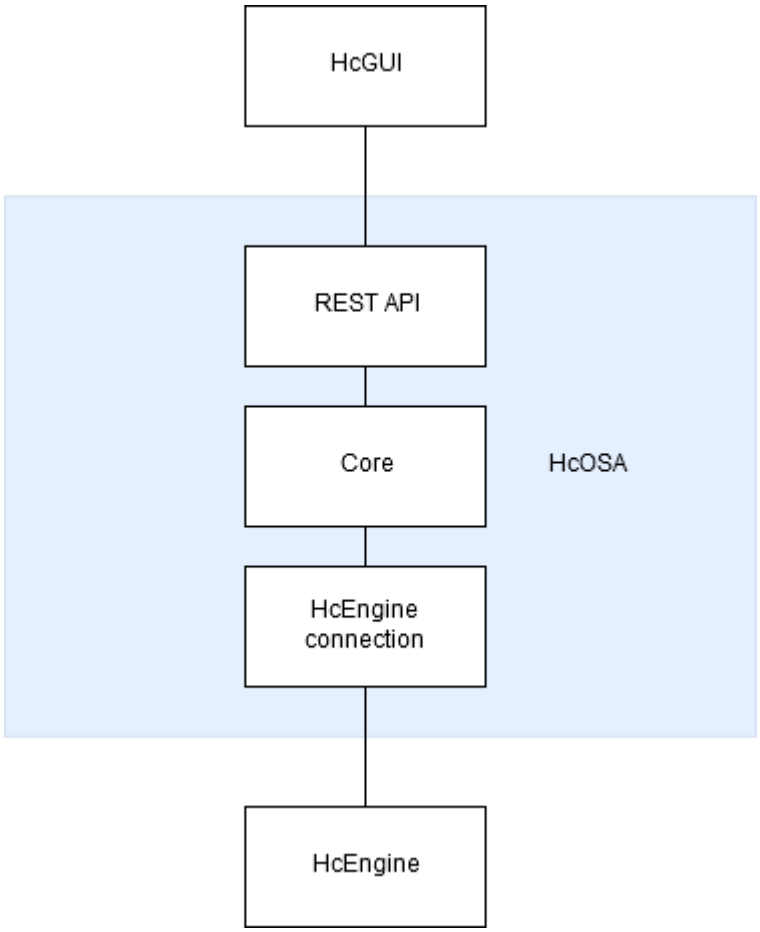
Tässä kappaleessa kuvataan suunnittelun eri vaiheita projektin alusta alkaen ja käydään läpi, kuinka uudet vaatimukset vaikuttivat ohjelmaan

4.1 Suunnittelun lähtökohta

Ohjelman suunnittelu ei alkanut minkään tietyn suunnittelumallin tai -periaatteen valinnalla. HcOSA:n yhtenä suurimpana vaatimuksena oli sen skaalautuvuus. Ohjelman tuli kyetä palvelemaan yhdestä moneen kymmeneen HcEngine-yhteyttä sekä ehkä jopa satoja käyttäjiä GUI:n välityksellä. Tämä skaalautuvuus ohjasi HcOSA:n suunnittelua alusta lähtien. Ohjelmiston pyrkimys toimia reaaliajassa myös tarkoitti, että operaatiot tuli suorittaa mahdollisimman tehokkaasti. Nämä olivat syitä, jotka johtivat ohjelmointikielen valinnassa C++:an. Diplomityön kirjoittajan kokemus kielestä sekä C++:n käyttö aiemmissa tiimin projekteissa vaikuttivat myös suuresti kielen valintaan.

Ohjelmalle sekä ohjelmistolle annettuja vaatimuksia oli vähän, ne olivat aika laajoja ja suunnittelulle taikka toteutukselle ei ollut tarjolla paljoakaan tarkkoja yksityiskohtia. Jotta ohjelmistosta saatiin parempi käsitys ja sen toimivuus todennettua, projektin alussa pääpaino oli saada ohjelmiston päästä päähän (end-to-end) tiedonkulku toimimaan. Ensimmäistä versiota HcOSA:sta ja HealthCheck-ohjelmistosta voitiin pitää prototyyppinä. Prototyyppiversiota ei ollut tarkoitus heittää roskiin sen valmistuttua vaan jatkaa kehitystä sen pohjalta. Projektia ohjelmistotuotantoprosessi voidaan mieltääkin evolutiiviseksi prototypisoinniksi (evolutionary prototyping). Tämänkaltaisessa prosessissa kehitetään ensiksi prototyyppi, jonka avulla kerätään palautetta jatkokehitystä varten. Tätä toistetaan, kunnes ohjelmisto on valmis. Vaikka prosessin mukaan kaikki ohjelmiston versiot ennen valmista versiota ovat prototyyppejä, miellettiin tämän projektin yhteydessä ainoastaan ensimmäinen versio varsinaisena prototyyppinä ja muut kehitysversioina.

HcOSA:lle annettujen vaatimusten mukaan, ohjelman pystyi jakamaan kolmeen selkeään pääosaan: rajapinta GUI:lle, logiikka ja rajapinta HcEngine:lle. Tekemällä GUI:lle tarkoitetusta rajapinnasta selkeä REST API, saataisiin täytettyä myös vaatimus rajapinnasta mahdollisille toisille ohjelmille. Prototyyppiversiossa HcOSA:n piti tulla toimeen ensiksi vain yhden GUI:n ja yhden HcEngine:n kanssa. Suunnittelussa keskityttiin aluksi tämän mahdollistamiseksi kuitenkaan unohtamatta tulevaa laajennusta useampiin yhteyksiin. Kuvassa 4 esitetään HcOSA:n sisäinen rakenne ja sen suhde muihin ohjelmiin.



Kuva 4. *HcOSA:n sisäinen korkean tason rakenne*

Jotta prototyyppi saatiin nopeasti valmiiksi, ei suunnittelulle jäänyt paljon aikaa. Suunnittelu ja toteutus etenivät rinnatusten ja usein toteutustekniset ongelmat ohjasivat suunnittelua. HcEnginen toteuttaja oli luomassa ohjelmalleen oman BNF-tyylisen syntaksikielen nimeltään Heccla. HcOSA käytti tätä kieltä HcEngine:n ohjauksessa, esimerkiksi kaavojen luomisessa ja kyselyiden tulosten keräämisessä. Tätä kieltä ei kuitenkaan haluttu käyttää suoraan GUI:n kanssa keskusteluun REST-rajapinnan selkeänä pitämisen vuoksi. Sen sijaan REST hyödyntää yleisesti käytettyä JSON-formaattia. Näiden rajapintojen eroavaisuuksien myötä HcOSA:n logiikkakomponentin tuli hoitaa viestien kääntäminen eri rajapinnoille sopiviksi.

HcOSA:n prototyyppiversion ei tullut suorittaa tuloksille minkäänlaista analyysia tai aggregointia. Tämän vuoksi HcOSA:n Logiikka-komponentilla ei ollut muuta virkaa kuin viestien muuntaminen ja välittäminen eri rajapintojen välillä. Tämäkin toiminnallisuus olisi pystytty sisällyttämään rajapintakomponentteihin minimalistisuuden vuoksi. Kuitenkin

jatkokehityksen kannalta oli tärkeää säilyttää komponentti rajapintojen välissä. Laskentalogiikan puutteen vuoksi komponentti nimettiin Core:ksi, sen ohjelman keskeisyyden vuoksi. Viestien välityksen ja muuntamisen lisäksi komponentille annettiin vastuu HcEngine-yhteyden hallinnasta. Core käynnistää ja sulkee yhteyden tarvittaessa ja tietää sen olemassaolosta. Tästä oli hyötyä jatkon kannalta, kun yhteyksiä oli useampia ja viestin välitys monimutkaistui.

C++ on olio-ohjelmointikieli ja siten olioihin pohjautuva suunnittelu oli selkeä valinta. Luokilla on selkeä kokonaisuus, jonka ne toteuttavat. Ideana oli, että prototyyppiversioon komponentit olisi muunnettu suoraan luokiksi, mutta toteutuksen yhteydessä kävi selväksi useampien luokkien tarve erityisesti rajapinnoissa.

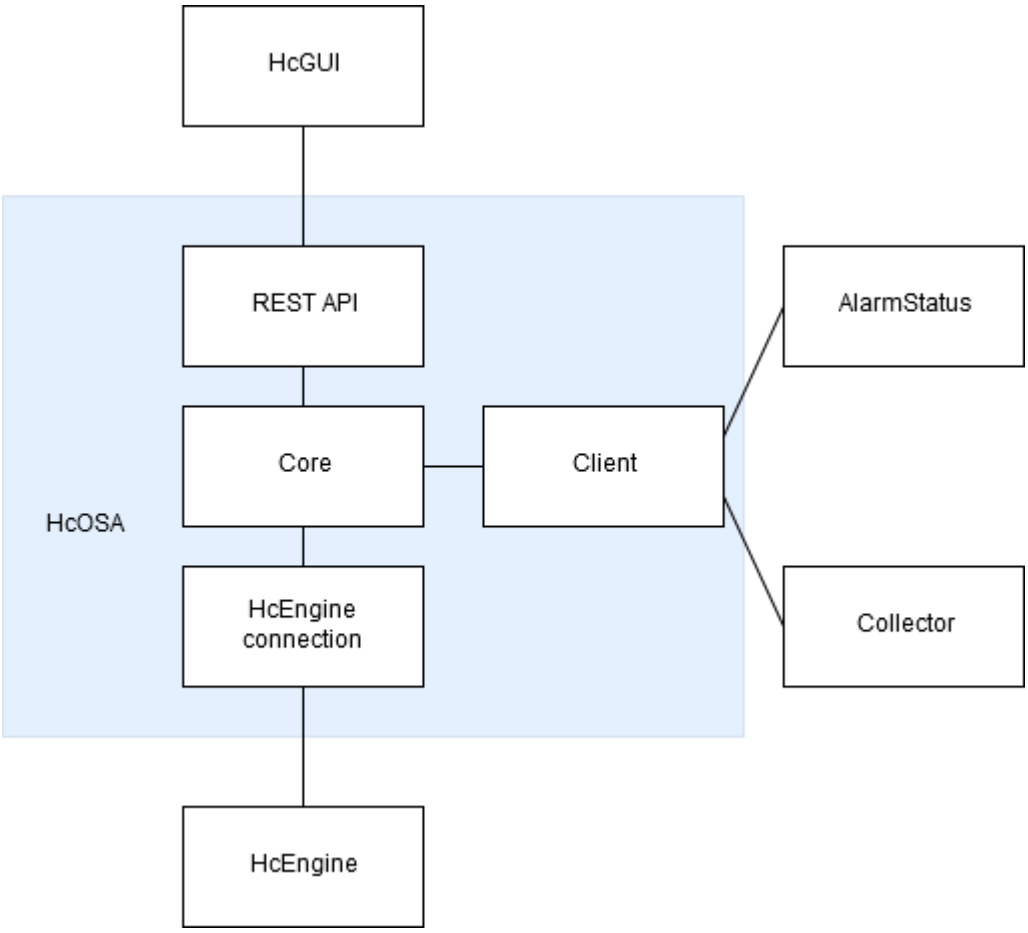
4.2 Jatkuva-aikainen suunnittelu

Ohjelmiston päästä päähän toiminallisuuden varmistuttua suunnittelun painopiste palasi takaisin usean HcEngine-yhteyden kanssa toimimiseen ja tehokkuuden optimoimiseen. Aiemmin tehty valinta Core-komponentin kyvystä hallita Engine-yhteyttä osoittautui oikeaksi. Se oli helppo laajentaa koskemaan useampia yhteyksiä. Nyt tuloksia tuli osata yhdistää useammalta HcEngine:ltä. Tämä tulosten yhdistäminen eli aggregointi olisi Aggregator-luokan vastuulla.

Ohjelmiston ensimmäinen version osoittauduttua erittäin hyvin toimivaksi ja siitä saadun palautteen takia, sille asetettiin lisävaatimus. Healthcheck-ohjelmiston tuli kyetä virhetilanteiden havaitsemisen yhteydessä käynnistämään kaiken kattava tiedon keräys virheen aiheuttaneesta tukiasemasta. Tämä edellytti HcOSA:lta kykyä muodostaa HTTPS-yhteys ja lähettämään keräilypyyntö Collector-ohjelmalle, joka oli vastuussa keräilyn aloittamisesta tukiasemasta. Keräilyn tulokset eivät palaudu HcOSA:lle vaan menevät suoraan tietokantaan. HTTPS-yhteyden muodostaminen ja asiakkaana toimiminen HTTPS-yhteyksien yli vaati uuden komponentin toteuttamista HcOSA:aan. Tätä kutsuttiin nimellä Client. HcOSA:n roolista monien erilaisten ohjelmien kanssa kommunikoinnista oli puhetta ja siten oli järkevää suunnitella tämä uusi Client-komponentti tukemaan monenlaisia yhteyksiä. Varsinainen Client-luokka oli vastuussa yhteyksien luomisesta ja viestien välityksestä oikealle yhteyden toteuttavalle luokalle.

Pian tästä ennakoivasta suunnittelusta oli hyötyä. HcOSA sai uuden ison lisävaatimuksen. Sen tuli pystyä tekemään ja lähettämään havaituista hälytystilanteista ilmoituksia AlarmStatus-ohjelmalle, joka oli muun muassa vastuussa matkapuhelinverkkojen hälytystilanteiden esittämisestä ja kirjanpidosta. Tämä vaatimus oli hyvin samanlainen korkealla tasolla tarkemman keräily -vaatimuksen kanssa. Toiselle ohjelmalle tuli välittää

tietoa HTTP tai HTTPS -protokollan yli. HcOSA:aan tuli lisätä vain uuden yhteystyypin toteuttava luokka Client-komponenttiin. Kuva 5 esittää HcOSA:n kasvanutta sisäistä korkean tason rakennetta ja sen suhdetta sekä HealthCheck-ohjelmiston toisiin ohjelmiin että ulkopuolisiin ohjelmiin.



Kuva 5. *HcOSA:n korkeantason rakenne ja ympäröivät ohjelmat*

Isompien vaatimusten lisäksi ohjelmistolle tuli pienempiäkin lisävaatimuksia. Nämä vaatimukset saattoivat koskea esimerkiksi tiettyä GUI:n näkymää tai tiettyä laskentakaavaa. Vaikka vaatimukset eivät suoraan olisi tulleet HcOSA:lle, vaikuttivat ne siihen rajapintojen välityksellä. Vaikka molemmat sekä GUI:n että HcEngine:n rajapinnat olivat suunniteltu prototyyppiversiota varten laajennettaviksi, oli ne kuitenkin tehty aika joustamattomiksi. Lisävaatimusten koskiessa olemassa olevia rajapintaelementtejä, niitä saatettiin joutua muokkaamaan aika rankastikin. Parin muutoksen jälkeen kävi selväksi, ettei jatkuva-aikainen korjaaminen olisi mitenkään järkevää vaan rajapinnat tulisi suunnitella joustaviksi ja helposti muutettaviksi.

Asiakkaiden tarpeiden yksityiskohdat voivat vaihdella ja siten oli tärkeää, että ohjelmiston toimintaa pystyttiin säätelemään halutun kaltaiseksi. Yhtenä perustavanlaatuisista lähtökohdista ohjelmistolle oli käyttäjän mahdollisuus luoda omia laskentakaavoja. Tämä mahdollistettiin GUI:ssa olevalla kaavaeditorilla. Jotta käyttäjien luomat kaavat olisivat käsiteltävissä ja laskettavissa, eivät kaavat voineet olla täysin mielivaltaisia vaan niillä oli oltava tietty syntaksi. HcOSA:n ja Engine:n vastuulla oli syntaksin oikeellisuuden varmistaminen. Erilaisia kaavoja kuitenkin pystyi olemaan käytännössä rajaton määrä, joten niiden kovakoodaaminen ei tullut kysymykseenkään, vaan kaavoja tuli käsitellä dynaamisesti. Usein käsittelyä pystyttiin helpottamaan jaottelemalla kaavoja niiden tuottamien tulosten formaatin mukaan. Tietyt kaavat voivat palauttaa vain yhden arvon, kun taas toiset viikkojen ajalta arvoja matriisissa.

Parametrit olivat toinen tapa mukauttaa ohjelmaa asiakkaalle sopivaksi. Näiden avulla tuli pystyä määrittelemään niin toisten ohjelmien sijainnit kuin kuinka herkästi HcOSA luo hälytyksen virhetilanteista. Ohjelmalle tuli pystyä käsittelemään parametrejä niin käynnistymisparametreinä kuin ajonaikaisparametreinä. Tämä tarkoitti sitä, että käynnistyksen yhteydessä parametrejä käytettiin alustamaan ohjelman toimintaa ja ajonaikana muuttamaan sitä. Näillä parametreillä oli vaikutusta kaikkiin ohjelman osiin ja siten ne luonnollisesti kuuluivat Core-komponenttiin, josta kaikki pystyivät hyödyntämään niitä.

5. TOTEUTUS

Tässä luvussa käydään läpi ohjelman toteutuksen eri vaiheita. Kerrotaan, miksi päädyttiin niin eri kolmannen osapuolen ohjelmiin kuin tiettyihin omiin ratkaisuihin.

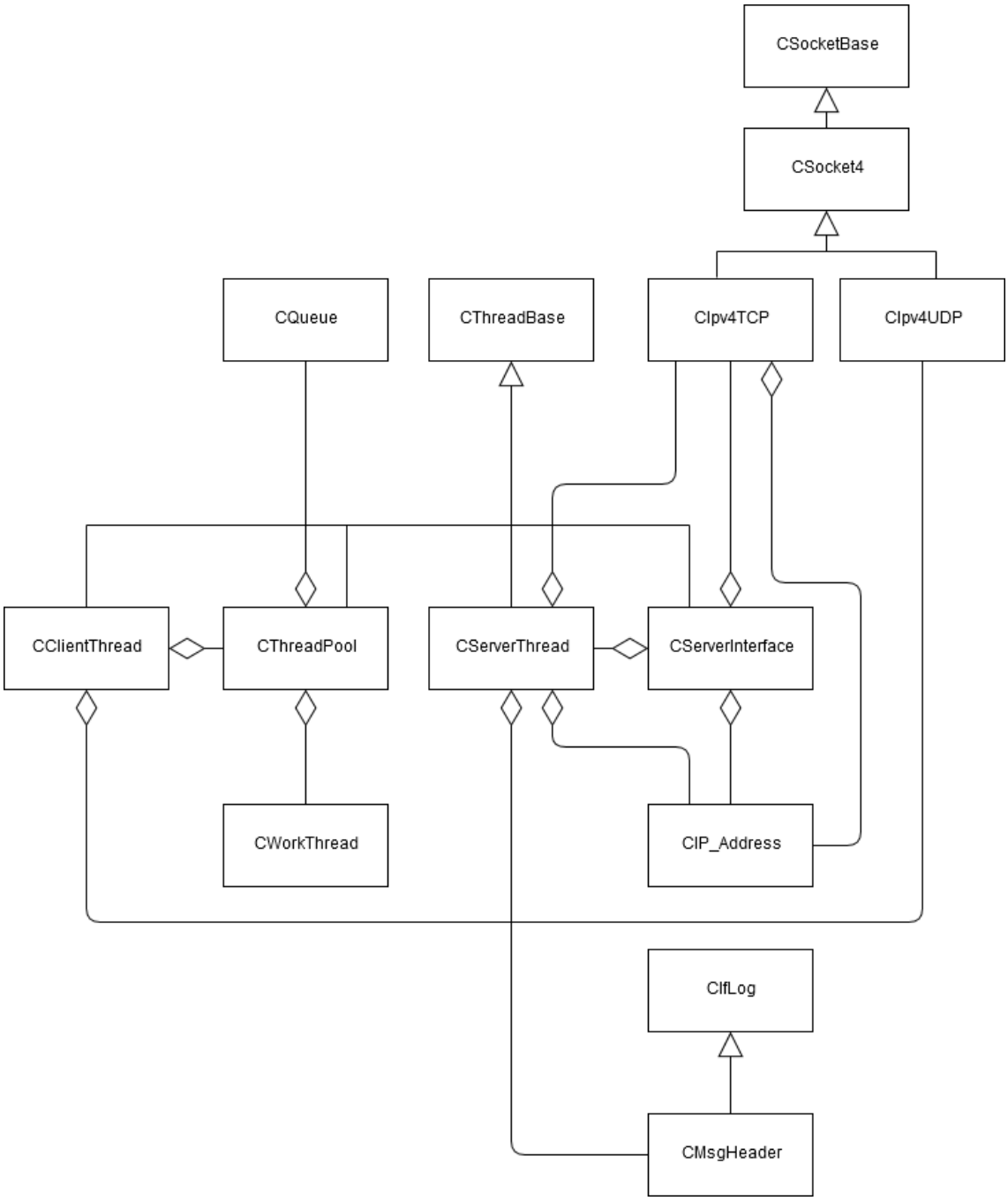
5.1 Aloituspohja

HcOSA:n koodipohjaksi käytettiin samaa pohjaa, mitä HcEngine hyödyntää. Tämä pohja on peräisin HcEngine-ohjelman toteuttajan edellisistä projekteista. Pohja sisältää pääpiirteissään luokat TCP ja UDP socketeille, näitä hyödyntävät server ja client luokat sekä lokinkirjoitusluokan. Kuvassa 6 nähdään koodipohjan yleisarkkitehtuuri ja luokkien nimet.

Saadut luokat loivat hyvän lähtökohdan HcEnginen kanssa kommunikointiin tarkoitetun rajapinnan toteutukselle. Yhteydessä käytettävän TCP-socketin toteutus vastasi hyvin pitkälti lopullista toteutusta. Ainoastaan viestien vastaanotosta vastaavaa funktiota jouduttiin muokkaamaan laajemmin. Jotta tiedetään HcOSA:n ja Engine:n välisen yhteyden olevan elossa, oli tärkeää lisätä ”vastaanotto”-funktioon heartbeat-käsite. Tämä tarkoittaa sitä, että mikäli funktio ei ole vastaanottanut mitään viestiä määritetyn ajan sisällä, lähetetään elokysely, heartbeat-viesti, HcEngine:lle. Mikäli tähän ei kuulu vastausta, yhteys katkaistaan.

HcOSA toimii clienttina ja HcEngine serverinä. Aloituspohjassa olevalle server-toteutukselle ei siis ollut suoranaista tarvetta. CServerInterface-luokan toiminta kuitenkin perustui yhteyksien hallintaan ja tästä saatiinkin muokattua CClientInterface-luokka, joka oli vastuussa yhteyksien luomisesta ja viestien välityksestä niille.

Aloituspohjasta löytyi myös lokikirjoituksen toteuttava luokka. Sitä käytettiin suuremmitta muokkauksitta HcOSA:n sisäisen toiminnan lokikirjoitukseen. Vaikkei lokikirjoituksesta ollut annettu vaatimuksia, uskottiin sen toteuttamisen olevan tärkeää ohjelman toiminnan seuraamisen ja vikojen etsimisen kannalta.



Kuva 6. *Koodipohjan yleisarkkitehtuuri.*

5.2 REST rajapinta

Alkuperäisiin vaatimuksiin nähden selkeä osa mikä puuttui aloituspohjasta, oli REST rajapinta GUI:lle. Vaatimuksena rajapinnalle oli viestien lähettäminen ja vastaanottaminen JSON-muodossa HTTP-yhteyden avulla. Tätä rajapintaa voidaan pitää REST-serverinä ja sen tuli pystyä palvelemaan useita samanaikaisia yhteyksiä. Prototyyppiversioon riitti kyky yhden samanaikaisen yhteyden käsittelyyn.

Toteutus aloitettiin tutkimalla valmiita C++:lle tarjolla olevia REST API-kehyksiä. Varteenotettavia vaihtoehtoja ei löytynyt kovin montaakaan. C++ REST SDK ja Pistache olivat käytännössä ainoat vaihtoehdot. WebSocket++ ja libcurl -kirjastojen avulla oli toki mahdollista toteuttaa oma REST API, mutta nämä eivät itsessään olleet valmiita ohjelmointikehyksiä ja nopean prototyypin kehityksen vuoksi nämä pystyttiin helposti hylkäämään.

C++ REST SDK on Microsoftin-projekti [8], joka tunnetaan myös sen alkuperäisellä nimellä Casablanca tai lyhenteellä Cpprest. Se pyrkii edesauttamaan C++-kehittäjiä toteuttamaan ja käyttämään RESTful-mallisia palveluja. Projekti hyödyntää asynkronista C++ API suunnittelua ja sisältää monia kirjastoja. Casablancan ominaisuuksiin kuuluu HTTP asiakas/palvelin, JSON, URI, asynkroniset striimit, WebSockets client ja oAuth. Ajoaikainen rinnakkaisuus toteutetaan Parallel Patterns Library (PPL) -kirjaston avulla. Vaikka projekti onkin Microsoftin kehittämä, on tuettuja ympäristöjä Windowsin lisäksi Linux, OS X, Unix, iOS ja Android. [5]

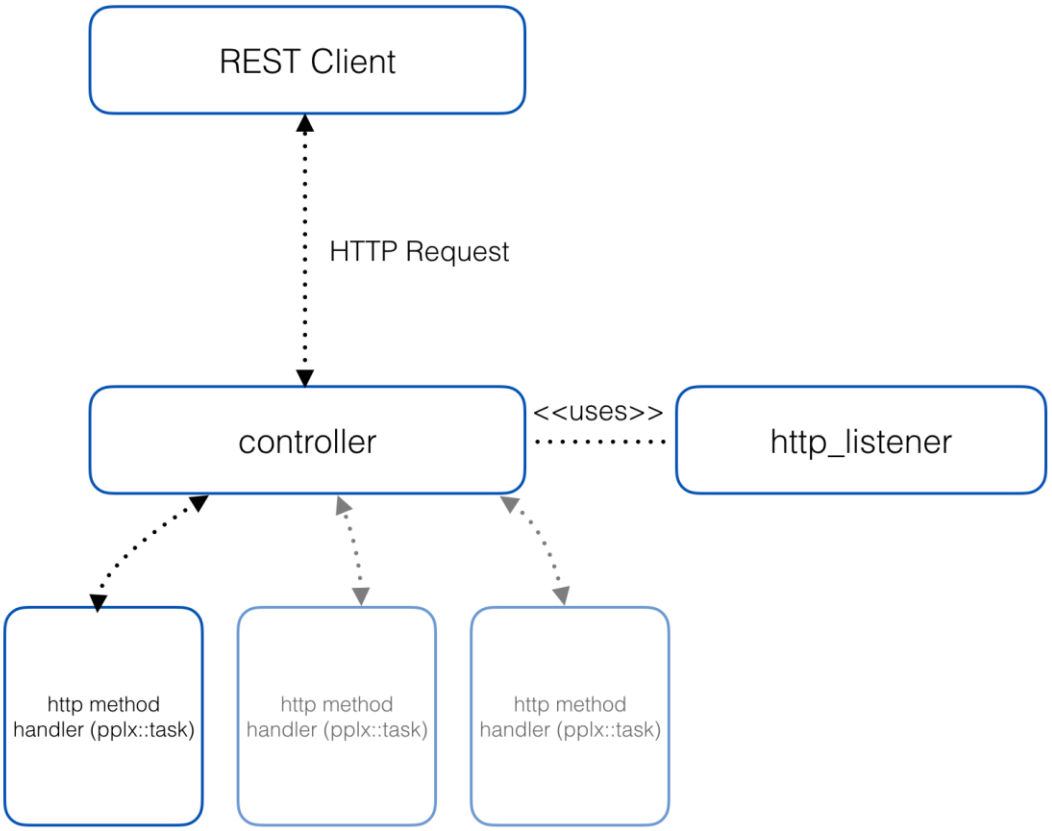
Pistache on Mathieu Stefanin kehittämä C++ REST framework [12], joka tarjoaa rinnakkaisuutta ja asynkronisuutta tukevan toteutuksen serverin ja asiakkaan API:ien rakentamiseen. Se on kevyt ja sillä ei ole riippuvuuksia toisiin kirjastoihin. Pistachea tuetaan myös OpenAPI:ssa, jonka avulla voi generoida annetulla päätepistekonfiguraatiolla valmiin aloituspohjan toteutukselle. Tämänkaltaisessa aloituspohjassa on luokat ja perusfunktiot luotu valmiiksi. Kuitenkaan varsinaista toiminnallisuutta ei ole.

Toteutukseen valittiin C++ REST SDK-kehys. Molempien rakenne oli selkeä ja modulaarisuutta tukeva sekä kumpikin tarjosivat rinnakkaisuutta tukevan mallin. Vaikka Pistachen OpenAPI tuesta on paljon apua nopeassa toteutuksessa, tuli ajatella pidemmälle. Suurena etuna Pistacheen nähden C++ REST SDK-projekti on Microsoftin kehittämä ja sillä on laaja käyttäjäkunta. Nämä tekevät projektin ylläpidon todennäköisemmin pitkäkestoisemmaksi ja apua saa ongelmatilanteissa luotettavammin.

Pohjana REST serverille toimi artikkeleissa [6,7] esitetty REST API rajapinta. Kirjoituksessa käydään läpi, kuinka Cpprest-kirjaston avulla voidaan toteuttaa asynkronisuutta tukeva REST serveri. Tämänkaltainen asynkroninen serveri pystyy palvelemaan useampia yhtäaikaisia yhteyksiä. Vaikka prototyyppiversioon ei tukea useammille yhteyksille vaadittu, oli tässä tapauksessa helpompi toteuttaa REST-serveri asynkroniseksi alusta lähtien, kuin muuttaa se jälkikäteen.

Kuva 7 esittää pohjan toimintaa. REST Client:lta tuleva HTTP-pyyntö saapuu controller:iin. Tämä käyttää pyynnön vastaanottamiseen http_listener-luokkaa, joka varmistaa

pyynnön oikeellisuuden ja varmistaa, että serveri tukee sitä. Tämän jälkeen pyyntö ohjataan sitä vastaavan eri threadissä toimivan http method handlerin käsiteltäväksi. Kukin näistä handlereistä toteuttaa yhden HTTP metodin (GET, POST, PUT…).



Kuva 7. *Artikkelissa esitetty REST serverin yleisarkkitehtuuri [6]*

Pohjassa esitetään ainoastaan yhden päätepisteen (endpoint) tukeminen. HcOSA:n REST serverin toteutuksen keskeisin osa on MainController-luokka. Tämä luokka on vastuussa tiettyihin päätepisteisiin kohdistuvien kyselyiden ohjaamisesta sopiville luokille. Jokaiselle päätepisteelle on oma MainController-luokasta periytetty luokkansa, joka toteuttaa tarvittavat GET, POST, PUT ja DELETE funktiot. Nämä Endpoint-luokat vastaanottavat pyynnön sisällön, muuttavat sen sopivaan muotoon, lähettävät sen Core-komponentille ja jäävät odottamaan vastausta. Saadun vastauksen perusteella muodostetaan paluuviesti asiakkaalle.

REST serverin asynkronisuus toteutetaan Parallel Patterns Library (PPL) -kirjaston ”tehtävä”-pohjaisella rinnakkaisuudella (Task Parallelism), koska C++ REST SDK-kirjasto käyttää hyväkseen samaa kirjastoa rinnakkaisuuden luomiseen. PPL on pelkistetysti kuvattuna rajapinta thread poolille. Kirjaston Task-luokan olioita käytetään suorittamaan

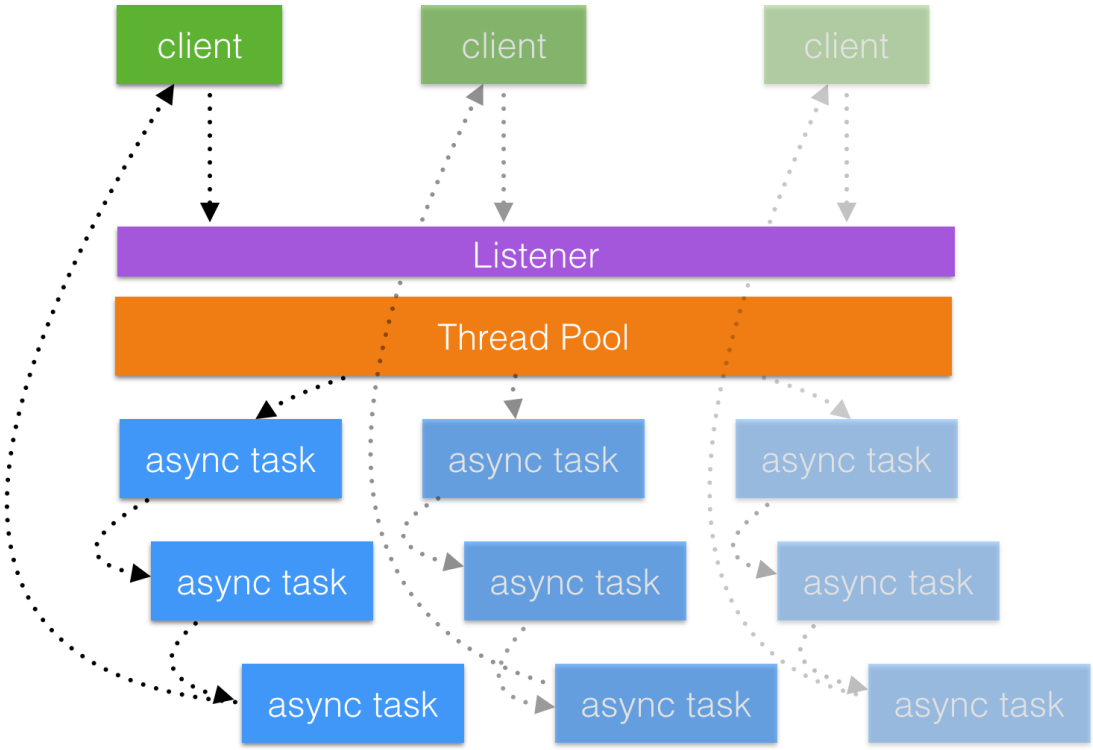
tehtäviä rinnakkain ja asynkronisesti. C++ REST SDK-kirjaston rinnakkaisuuden toteutus käsittelee monia yhtäaikaisia yhteyksiä. REST serverin toteutuksessa asynkronisuutta voidaan lisätä ketjuttamalla Task-olioita toisiinsa eli toinen task aloittaa suorittamaan omaa osuuttaan toisen valmistuttua. Ohjelmassa 1 esitetään käytettyä tehtävien ketjutusta. Ensimmäinen task suorittaa GUI:lta tulleen viestin sisällön purkamisen ja välittämisen Core-komponentille. Tämän jälkeen ketjutetaan toinen task hoitamaan Core:lta saadun vastauksen parsiminen GUI:lle lähetettävään muotoon. Ketjutuksen ansiosta ensimmäinen threadi voidaan vapauttaa palvelemaan uusia pyyntöjä pitäen REST serveri nopeasti reagoivana. [9]

```
// Task extracts json from the message
message.extract_json(true).then([ &message, this]
(pplx::task<json::value> task) {
    // Body of the request
    auto obj = task.get();
    auto formula_name = obj.has_field("formula_name") ?
         obj.at(U("formula_name")) : json::value::null();
    auto formula_body = obj.has_field("formula_body") ?
         obj.at(U("formula_body")) : json::value::null();

    // Message is sent to core and answer received
    string answer = core->CreateFormula(formula_name, formula_body);
    // Next task is created to handle parsing and sending the response
    task.then([ &message, &answer, this](pplx::task<json::value> task) {
        // Response is formed
        json::value::object response = ParseResponse(answer);
        // Response is sent
        message.reply(status_codes::OK, response);
    }).wait();
}).wait();
```

Ohjelma 1. *Task:ien avulla luodun asynkronisuuden hyödyntäminen REST rajapinnassa formulan luomisessa.*

REST serveri on toteutettu siten, että lähettäessään viestin GUI jää odottamaan vastausta siihen. Vastauksen saapumisessa voi kestää pitkäkin aika ja joissakin REST servereissä tämä johtaisi ongelmiin, koska odotuksen aikana ei voitaisi palvella muita pyyntöjä. Kuvan 8 mukaisesti toteutetussa REST serverissä muiden pyyntöjen palvelu onnistuu odotuksenkin aikana toteutuksen rinnakkaisuuden ja asynkronisuuden vuoksi. Toteutuksessa MainController-luokka ohjaa kuvassa Listerner:n vastaanottamat viestit ja ohjaa ne kuvan Thread Pool:ia hyväksikäyttäen oikeille Endpoint-luokille, jotka käsittelevät viestin asynkronisien task-olioiden avulla.



Kuva 8. *Malli REST serverin rinnakkaisuutta tukevasta rakenteesta [7]*

Jos odotusajat kasvavat jatkossa liian pitkiksi, voidaan serverin toteutusta muuttaa siten, että vastauksen odottamisen sijaan GUI:lle palautetaan uniikki tunnistekoodi ja pyydetään GUI:ta kysymään vastausta myöhemmin tunnistekoodin perusteella. Tällöin GUI ei tee yhtä pitkäkestoista kyselyä, vaan useampia nopeampia kyselyitä, kunnes vastaus on valmis.

5.3 Prototyyppi

Ohjelmiston ensimmäinen versio oli prototyyppi ja sen tuli tukea vain perustoimintoja ohjelmiston idean todentamiseksi. HcEngine:n suorittamaan laskentaan liittyy koko ohjelmiston kannalta kolme oleellista käsitettä: kaava (formula), laskenta (computation) ja tulos (result). Kaavan avulla kerrotaan, mitä arvoja kaavan tulee laskea, sen tuloksen tyyppi, tarkkuus ja arvoväli. Kaava yksinään ei tee vielä mitään, vaan vasta laskennan asetuksen myötä HcEngine alkaa keräämään tarvittavia arvoja. Laskennassa annetaan käytetyn kaavan nimi, laskennan kohdejoukko ja arvojen keräystarkkuus. Näiden lisäksi voidaan antaa lisämääreitä, kuten virhejakauman kerääminen. Laskenta johtaa ainoastaan kaavan laskemiseen tarvittavien arvojen keräämiseen. Vasta kun laskentaan kysytään tulosta, laskee HcEngine sen käyttäen keräämiään arvojaan. Tuloksen kysymisen yhteydessä voidaan antaa tarkennuksia, esimerkiksi miltä ajalta tai tukiasemilta tulos

halutaan tai halutaanko yksittäinen tulos vai tulokset kaikilta aikajaksoilta. Tiivistäen, HcEngine:llä on varastossaan useita kaavoja, joita voidaan käyttää erilaisissa laskennoissa. Kun laskenta aktivoidaan tietylle kaavalle, aletaan keräämään arvoja tämän laskennan suorittamiseksi. HcEngine suorittaa laskennan vasta sitten, kun siltä kysytään tulosta siihen. Prototyyppiversion ei tarvinnut tukea laskennan määrittelemisen ja tuloksen kysymisen yhteydessä annettavista lisämääreistä.

HcOSA:ssa REST ja HcEngine -rajapintojen tuli tukea vain yhtä yhteyttä. REST serveri saatiin kuitenkin toteutettua asynkroniseksi niin pienellä vaivalla, että ei koettu siitä koituvan lisätyön ylittävän saatua hyötyä jatkokehityksen kannalta. HcEngine-yhteyden kanssa toimittiin hieman samalla tavalla. Helpoin tapa olisi ollut staattisesti yksi yhteys, jolle kaikki liikenne ohjataan. Jatkon kannalta oli tärkeää pystyä kuitenkin hallitsemaan yhteyksiä ja pystyä jakamaan niille erilaisia tehtäviä. Tämän takia prototyyppiversioon toteutettiin hyvin yksinkertainen yhteyksien hallitsijaluokka. Tässä vaiheessa luokan toiminnallisuus oli käytännössä yhden yhteyden luominen ja sille viestien välitys. Jatkon kannalta tätä oli kuitenkin helppo laajentaa koskemaan useampia yhteyksiä.

Asynkronisuuden sijasta REST serverissä jätettiin toteuttamatta vähemmän tärkeät päätepisteet, kuten parametreihin ja statistiikkaan liittyvät päätepisteet. Prototyyppiversiossa keskityttiin toteuttamaan oleellisimmat päätepisteet, kuten kaava, laskenta ja tulos. /formula-päätepisteen kautta GET-metodilla sai kaikki käytössä olevat kaavat, POST:lla sai luotua uusia kaavoja ja DELETE:llä poistettua. Vastaavasti /computation-päätepisteestä sai kaikki suorituksessa olevat laskennat sekä luotua uusia ja poistettua olemassa olevia. Tulos saatiin /result-päätepisteen POST-metodilla. Viestin yhteydessä lähetettiin laskennan nimen lisäksi lisämääreitä. Annetuiden määreiden pohjalta laskettiin ja muotoiltiin tulos halutunlaiseksi. Prototyyppiversioon näihin kaikkiin päätepisteisiin tehtiin helpotuksia erityisesti GUI:lta tulevan viestin käsittelyyn. Kovakoodatut viestit lähetettiin valmiiksi siinä muodossa kuin HcOSA käsittelee niitä sisäisesti, jolloin ei ollut tarvetta tulkita ja muuntaa viestiä. Osittain tämä myös johtui siitä, että tällöin GUI:nkin osuus viestien luomisessa helpottui.

REST serveriltä viestit välitettiin tässä vaiheessa suoraan EngineInterface-luokan oliolle. Se muodosti saadusta datasta halutun kaltaisen Heccla-muotoa olevan viestin ja välitti tämän EngineConnection-yhteys oliolle, joka sitten edelleen lähetti sen HcEnginelle. Vastauksen tullessa EngineConnection-olio talletti oleellisen osan ja välitti sen EngineInterface-luokalle, joka taas palautti sen REST serverille. Täällä se muokattiin GUI:lle sopivaan muotoon ja lähetettiin sille takaisin.

5.4 Toteutuksen jatkaminen

Prototyyppiversion valmistuttua kehityksessä siirryttiin toteuttamaan ja laajentamaan poisjätettyjä sekä prototyypin kehityksen ja siitä saadun palautteen perusteella opittuja ominaisuuksia. REST serveriin lisättiin päätepiste-elementtejä ja HcGUI:lta tulevien viestien käsittely muunnettiin dynaamiseksi tukemaan käyttäjien itse luomia kaavoja. Liite A:sta löytyy HcOSA:n REST API:n kuvaus. Ohjelmaan myös lisättiin rajapintojen lokikirjoitus ja ohjelman ohjaukseen käytettävät parametrit.

Suurin työ oli saada HcOSA toimimaan monien HcEngine:iden kanssa. Prototyyppiin oli tehty valmiiksi hyvät perustat useiden yhteyksien hallinnalle ja sitä laajentamalla saatiinkin useampi yhteys käyntiin helposti. Vaikein osuus oli kuitenkin vastauksien saaminen useammasta lähteestä ja niiden yhdistäminen yhdeksi REST:lle annettavaksi viestiksi. Jokaista yhteyttä vastasi yksi EngineConnection-luokan olio. Nämä toimivat täysin toisistaan riippumattomasti. Vastauksen tullessa ne välittävät sen EngineInterface-oliolle. Saatuaan kaikilta EngineConnection-olioilta vastauksen EngineInterface-olio antaa vastuun vastausten yhdistämisestä Aggregator-oliolle. Erilaisia viestejä tulee yhdistää eri lailla. Esimerkiksi kysyttäessä HcEngine:n käytössä olevia kaavoja, voidaan muodostaa lista, jossa kaava on kertaalleen ja toisessa kentässä kerrotaan missä kaikissa laskentayksiköissä se on käytössä.

Laskentatulosten kanssa tulee taas menetellä eri tavalla. Tuloksia ei voi vain suoraan yhdistää ja olettaa, että saatu vastaus olisi oikea. HcEngine:t laskevat vain osan matkapuhelinverkon koko liikenteestä ja verkon kokonaistilan laskemiseksi tulee näiden laskentaohjelmien tulokset yhdistää. Kyseessä ei kuitenkaan ole yksinkertainen yhteenlasku, sillä verkon liikenne ei todennäköisesti ole jakautunut tasaisesti eri laskentaohjelmien välillä. Oletetaan, että HcOSA:lla on kaksi HcEngine-yhteyttä ja lasketaan puheluiden onnistumisprosenttia. Engine 1 palauttaa arvon 95% ja Engine 2 palauttaa 99%. Nopeasti ajateltuna oikea vastaus olisi näiden keskiarvo eli 97%. Kuitenkin HcEngine:iden alueilla tapahtuvien puheluiden määrä voi vaihdella merkittävästi. Tässä tapauksessa Engine 1:n alueella tapahtui 100 puhelua ja toisen alueella 1000 puhelua. Nyt voidaan laskea oikea tulos koko alueelle, mikä tässä tapauksessa on 98,6%.

Prototyyppivaiheessa HcEngine palautti tulosten yhteydessä ainoastaan tukiaseman tunnisteen ja tuloksen arvon prosentteina. Jotta HcOSA pystyi yhdistämään eri HcEngine:iltä tulevat tulokset, tuli laskentayksiköiden palauttaa tulosten lisäksi laskennassa käytetyt arvot. Näiden avulla ohjelma sai laskettua uudet, koko verkkoa vastaavat arvot. Tässä vaiheessa kaikki HcEngine:n suorittama laskenta oli kohtalaisen yksinkertaista ja

laskut pystyttiin yleistämään jakolaskuksi, esimerkiksi onnistuneiden määrä jaettuna yritysten määrällä.

5.5 Muutokset ja lisävaatimukset

Prototyypin jälkeen rajapintojen sisältöjä ei vain laajennettu vaan niitä jouduttiin myös muuttamaan. HcEngine:ssä käytetty Heccla-kieli oli jatkuvan kehityksen alla ja usein siinä tapahtuvat muutokset vaikuttivat HcEngine:n lisäksi HcOSA:aan. Suurimmat muutokset saattoivat jopa heijastua HcOSA:n läpi aina REST serverin kautta HcGUI:hin. Myöskään kieleen tehtävät lisäykset eivät pelkästään johtaneet uuden toteutuksen luomiseen vaan mahdollisesti vaikuttivat jo olemassa oleviin rajapintoihin ja funktioihin. Esimerkiksi 5G:hen liittyvien laskentojen lisäys tuli huomioida laskentatulosten vastaanotossa HcOSA:ssa. 5G NSA (non-standalone) käyttää sekä 4G että 5G tukiasemia ja laskentatuloksissa näkyy näiden molempien tukiasemien tunnisteet. Tätä ennen HcOSA oli odottanut ainoastaan yhtä tukiaseman tunnistetta laskutulosta kohden.

Prototyypin osoittauduttua onnistuneeksi luottamus ohjelmistoon kasvoi ja sille asetettiin lisävaatimuksia. HcEngine havaitessaan virhetilanteen tai poikkeaman luo ja lähettää hälytyksen HcOSA:lle. Ensimmäisen lisävaatimuksen mukaan HcOSA:n tuli hälytyksen pohjalta aktivoida kattavampi keräily suoraan tukiasemasta. Keräilystä vastasi toinen ohjelma nimeltään Collector. HcOSA:n tehtävä oli lähettää tälle ohjelmalle HTTPS-viesti, jossa kerrottiin tarvittavat tiedot keräyksen aloittamiseen, kuten hälytyksen aiheuttaneen tukiaseman IP-osoite. Collector-ohjelma sitten otti yhteyttä tukiasemaan ja aloitti keräyksen, jonka tulokset kirjoitettiin levylle. HcOSA välittäisi GUI:lle tiedon mistä keräilytulokset olivat löydettävissä.

HcOSA:n ei ollut ennen tätä vaatimusta tarvinnut toimia REST-asiakkaana ja tämän vuoksi siihen tuli toteuttaa koko Client-komponentti. Ideana oli käyttää C++ REST SDK tarjoamaa toteutusta REST clientille, mutta HTTPS-tuki osoittautui ongelmalliseksi. Tämän takia toteutuksessa käytettiin LibCurl-kirjastoa sen tarjoaman selkeän SSL-sertifikaatti- ja HTTPS-tuen takia. LibCurl on matalamman tason kirjasto eikä suoraan tarjoa REST client toteutusta, mutta se antaa erittäin laajat ja tarkat vaihtoehdot sen rakentamiselle. Tästä oli apua, sillä viesteissä piti huomioida sertifikaattien lisäksi myös autentikointi ja tarkka viestin rakenne.

Client-komponentista haluttiin rakentaa helposti laajennettava, koska oli hyvin mahdollista, että HcOSA:n tuli lähettää viestejä muillekin ohjelmille jatkossa. Erilaisia yhteyksiä

hallitsemaan luotiin MainClient-luokka, joka toimi rajapintana erilaisille yhteyksille ja vastasi sisällön välityksestä niille. Yhteyden toteuttava luokka määritteli viestin rakenteen ja vastaanottajan.

Toinen lisävaatimus koski hälytysten lähettämistä hälytyksistä vastaavaan keskusohjelmistoon. HcOSA:n tuli muodostaa ilmoitus HcEngine:ltä tulevista hälytyksistä ja lähettää se hälytysohjelmalle. Ilmoituksen lähetys vastasi jonkin verran kattavan keräilyn aktivointia ja lähetyksen toteuttavan luokan lisäys Client-komponenttiin ei ollut suurikaan vaiva. Eniten aikaa kului ilmoituksen syntaksin hiomiseen ja sen rekisteröimiseen hälytyskeskusjärjestelmään.

5.6 Rinnakkaisuus

Luvussa 5.2 käsiteltiin REST serverin rinnakkaisuuden toteutusta. Muussa osassa ohjelmaa rinnakkaisuus toteutettiin C++:n standardikirjastoon kuuluvalla säiekirjastolla (Thread Support Library). Kirjaston säietoteutus on alustariippumaton ja siten sen tehokkuus on paras, mitä käytössä oleva käyttöjärjestelmä pystyy tarjoamaan. Vaikka kirjaston tarjoama asynkronisuus ja säikeiden luominen ei ole parhaimmasta päästä, ei niillä ole merkitystä tässä tapauksessa, sillä ohjelmassa ei käytetä asynkronisuutta ja säikeet ovat pitkäikäisiä. Tärkeää on, että säiekirjasto kuuluu standardikirjastoon ja siten ohjelmaan ei tule ylimääräisiä riippuvuuksia toisiin kolmannen osapuolen kirjastoihin. Kirjasto on myös selkeä ja helppokäyttöinen. [14]

Ohjelman ensimmäinen säie luodaan hallitsemaan HcEngine-yhteyksiä. Säikeessä suoritettava funktio huolehtii, että HcOSA:lla on yhteydet kaikille sille annettuihin HcEngine:ihin. Mikäli niitä on liian vähän tai liian paljon, käynnistää tai sammuttaa se yhteyksiä. Tämä säie on käynnissä koko ohjelman ajan. Yhteyden käynnistyessä EngineConnection-oliossa luodaan säie, joka on vastuussa HcEngine:ltä saapuvien viestien lukemisesta suoraan socketista. HcOSA:n ja HcEngine:n välillä kulkevissa viesteissä on otsikko-osio, jossa kerrotaan muun muassa viestin tyyppi ja sen pituus. Viestin pituus voi olla niin suuri, ettei se mahdu yhteen TCP-pakettiin, minkä maksimikoko on 64 KiB. Tällöin säikeen vastuulla on koota paketit yhteen, kunnes kasassa on otsikossa ilmoitetun viestin pituus. Matkalla voi TCP-paketille tapahtua myös fragmentaatio, sillä yleisesti käytetty Ethernet tukee ainoastaan 1500 tavun pituisia TCP-paketteja [10]. Socketissa hyödynnetään kuitenkin kernel-tason toteutusta pakettien vastaanotossa, joka hoitaa pilkkoutuneiden pakettien kokoamisen yhteen automaattisesti.

Hälytyksen tullessa lähetetään se hälytyskäsittelijälle. Tämä on käytännössä tuottaja-kuluttaja-mallinen (producer-consumer) ratkaisu. Viesti kirjoitetaan kaksipäiseen jonoon

(std::deque) ja lähetetään ilmoitus säikeelle, joka on joko nukkumassa tai suorittamassa aiempaa hälytystä. Säikeen tehtävä on tutkia hälytystä ja päättää mitä sille tulee tehdä. Päätöksen teon avuksi saatetaan jatkossa kysyä apua toisilta ohjelmilta. Tämänkaltainen kysely on kuitenkin todennäköisesti hidas ja vastauksen odotus pysäyttäisi koko säikeen toiminnan. Tämän vuoksi säie päätettiin jakaa kahteen osaan. Ensimmäinen säie on vastuussa hälytyksen sisällön oikeellisuuden varmistuksesta ja tarvittaessa tekee kyselyn aloituksen toiselle ohjelmalle. Tämä ohjelma käynnistää tarvittavan tiedon haun ja palauttaa HcOSA:lle saman tien uniikin tunnistekoodin, jonka avulla haun tulokset voidaan kysyä sen valmistuttua. Säie sitten kirjoittaa tämän tunnistekoodin hälytyksen kanssa toiseen kaksipäiseen jonoon, josta toinen säie lukee ne. Tämä säie on vastuussa päätöksestä mitä hälytykselle tulisi tehdä. Päätökseen vaikuttaa sekä sisäiset tiedot että mahdollisesti toisilta ohjelmilta kysytyt aputiedot. Säie pyytää vastausta toiselta ohjelmalta sille annetun tunnistekoodin avulla. Mikäli vastaus ei ole vielä valmis, kysytään hetken päästä uudestaan ja tätä jatketaan, kunnes vastaus saadaan taikka tietty aika ylittyy. Tämän jälkeen säie tekee päätöksen jatkotoimenpiteistä hälytyksen suhteen. Esimerkiksi se voi lähettää hälytyksen tiedot toisessa säikeessä pyörivän Client-komponentin jonoon odottamaan tarkemman keräyksen aloittamista tukiasemassa. Hälytyksen käsittelijän jakaminen kahteen osaan ei auta tilanteissa, joissa hälytyksiä tulee yksitellen. Mutta jos niitä useampia, pystyy ensimmäinen säie aloittamaan kyselyt muissa ohjelmissa ajoissa, jolloin toinen säie ei joudu odottamaan ainakaan niin kauaa vastauksen saamista ensimmäisen hälytyksen jälkeen.

5.7 Julkaisuversio ja ohjelman toiminta

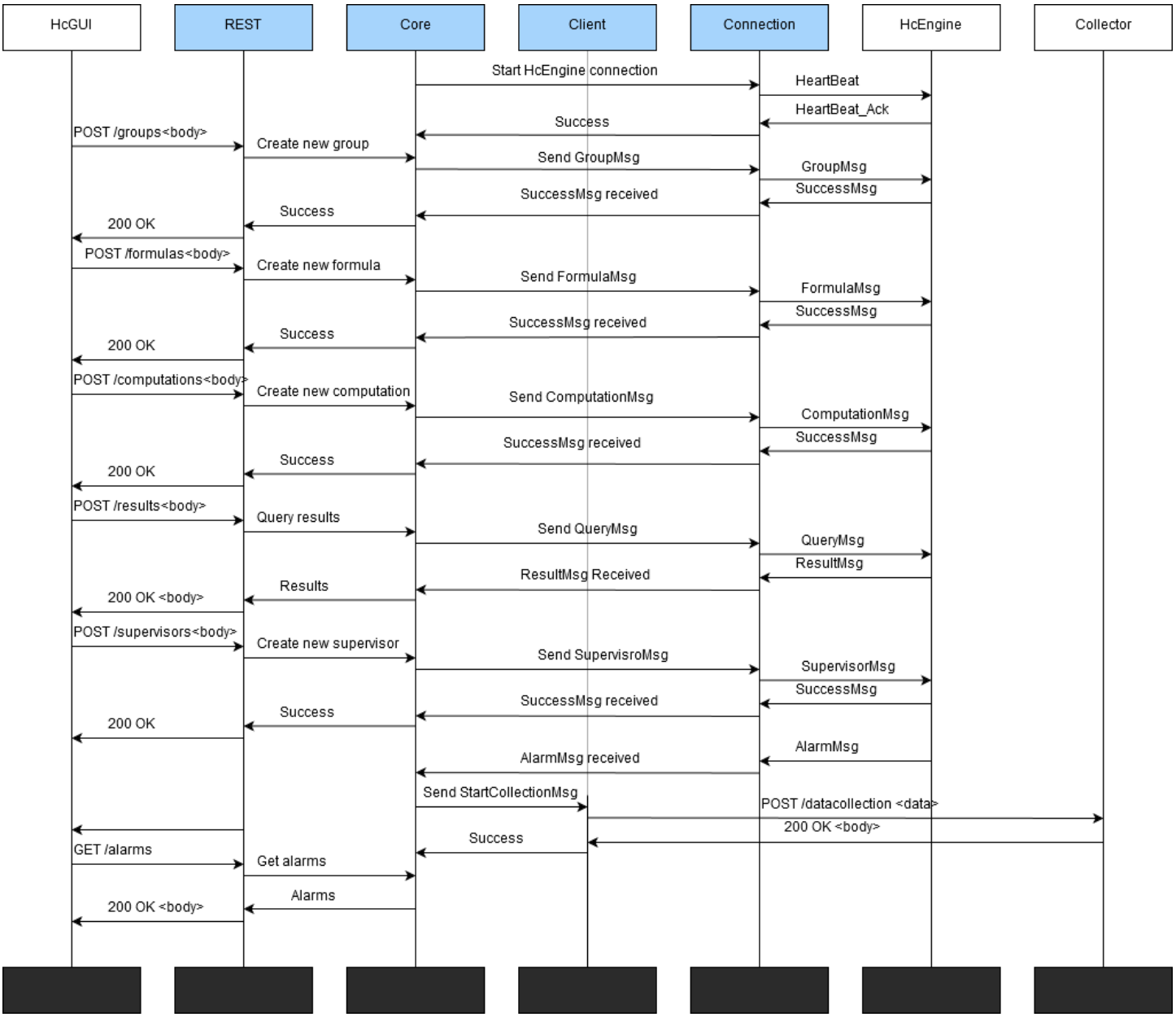
Yrityksen sisäisten tavoitteiden muuttumisen myötä HealthCheck-ohjelmiston ensimmäiset julkaisuversiot olivat tarkoitettu ainoastaan pieniin, yhden reunaserverin alaisuudessa toimiviin, verkkoihin. Tämä tarkoitti sitä, että HealthCheck-ohjelmistossa oli ainoastaan yksi HcEngine. HcOSA:n roolin pääpaino siirtyi eri HcEngine:iltä tulevien viestien yhdistäjästä hälytysten analysoijaksi ja jatkotoimenpiteiden päättäjäksi.

HcOSA:n toteutuksessa otettiin edelleen huomioon laajennettavuus useiden HcEngine:iden kanssa toimimiseen ja aikataulun salliessa myös toteutettiin se. Mutta esimerkiksi HcEngine:n laskennan kehityttyä, ei HcOSA pystynyt yhdistämään uusien monimutkaisten laskentakaavojen tuloksia useammalta HcEngine:ltä ja sen toteuttaminen olisi vienyt liikaa aikaa. Täten ensimmäisessä julkaisuversiossa HcOSA ei täysin tukenut tulosten yhdistämistä ja tästä johtuen useamman HcEngine:n tukeminen oli vaillinaista. Tämän korjaamisesta enemmän luvussa 7.6 Jatkokehitysideat.

Ensimmäiseksi ohjelman käynnistyessä tutkitaan sekä käynnistysparametrit että konfiguraatiotiedoston parametrit ja toimitaan niiden mukaan. Nämä parametrit muodostuvat käytännössä sekä muuttujan nimestä että arvosta ja niillä ohjataan pääasiassa ohjelman käynnistymistilaa ja konfiguraatioarvoja. Mikäli käynnistymisparametri ja konfiguraatioparametri koskevat samaa muuttujaa, huomioidaan ainoastaan käynnistymisparametrin arvo. Kun lähtöarvot ovat luettu ja tarvittavat muutokset tehty, luodaan rajapinnat ja muodostetaan yhteydet tarvittaviin muihin ohjelmiin. Yhteydet testataan ja tulokset kirjoitetaan komentoriville ja lokitiedostoon. Käynnistymisen jälkeen ohjelman ohjaus tapahtuu pääasiassa REST rajapinnan kautta annettavilla komennoilla tai suoraan konfiguraatiotiedostoa muuttamalla.

Kun yhteydet on muodostettu, ohjelman toiminta voi alkaa. Yksinkertaisimmillaan ohjelma odottaa kyselyjä REST rajapinnasta, muuntaa nämä laskentakomponentille sopivaan muotoon ja lähettää ne sille. Vastauksen tullessa takaisin, sen sisältö tutkitaan ja mikäli se ei johda mihinkään lisätehtäviin muunnetaan se takaisin pyydettyyn muotoon ja vastataan alkuperäiseen REST rajapinnasta tulleeseen kyselyyn. Tässä tapauksessa analysointiohjelma toimii käytännössä viestin muuntimena ja välittäjä.

Toimenkuva mutkistuu huomattavasti, jos viestiä tulee tutkia tarkemmin. HcEngine:lle voidaan antaa sääntöjä ja raja-arvoja, joiden ylittyessä lähetetään hälytys HcOSA:lle. Sen vastuulla on tutkia hälytystä ja päättää mitä sille tulee tehdä. Mikäli todetaan, että tämä hälytys on oikeanlainen, HcOSA voi ohjeistaa toisen ohjelman aloittamaan erittäin tarkan tiedon keräyksen hälytyksen aiheuttaneeseen tukiasemaan. Kerätty tieto tallennetaan levylle, josta se on myöhemmin tarkasteltavissa. Hälytyksistä annettaan erilaisia ilmoituksia GUI:lle riippuen niiden tyypistä ja mitä niille tehtiin. Jos hälytys on johtanut tiedon keräykseen suoraan tukiasemasta, kerrotaan mistä kerätty tieto löytyy mahdollista jatkokäsittelyä varten. Kuvassa 9 on HcOSA:n MSC-kaavio (Message sequence chart). Kaavio kuvaa HcOSA:n kannalta oleellisimpien viestien liikkumista eri ohjelmien ja komponenttien välillä.



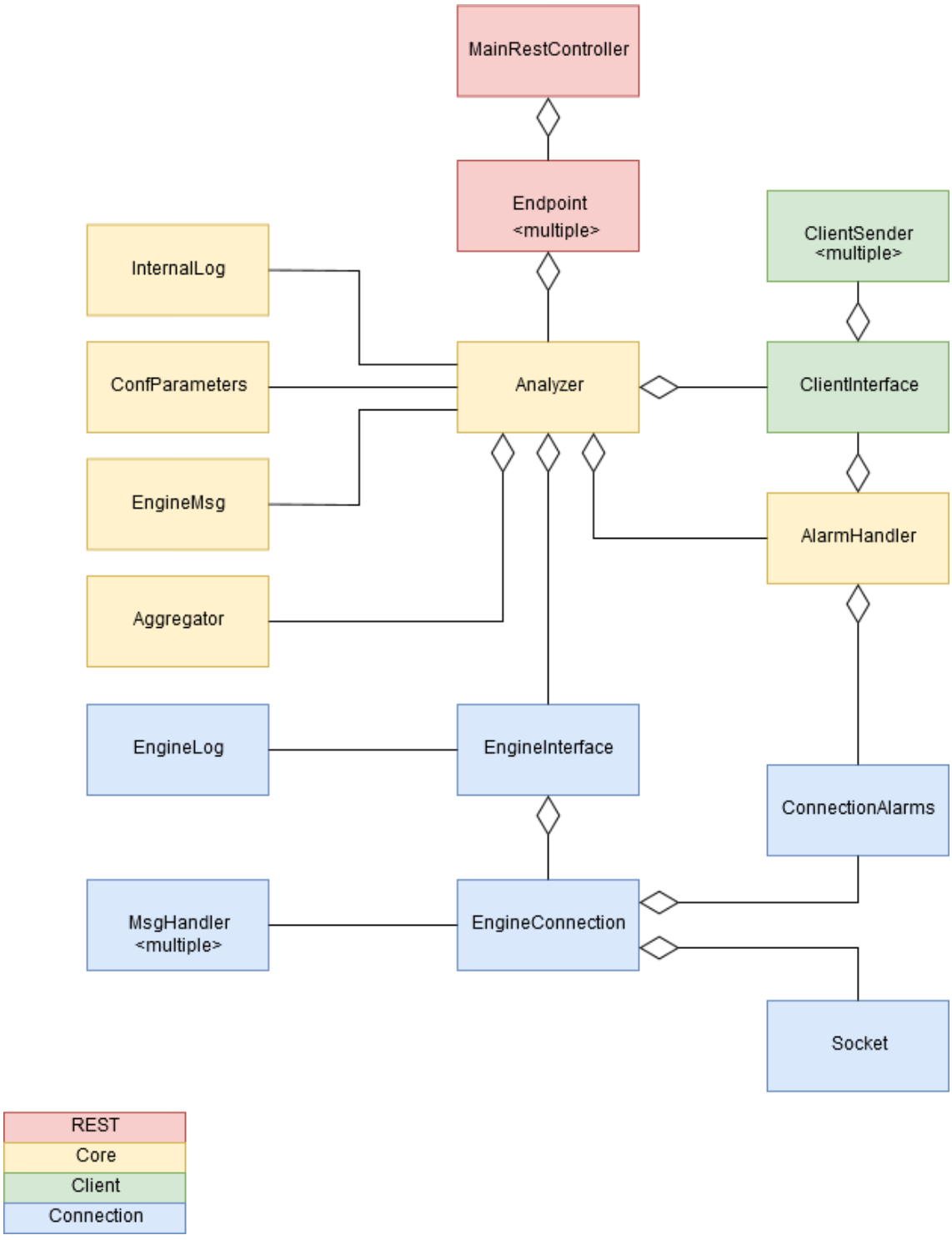
Kuva 9. *HcOSA:n MSC-kaavio. HcOSA:n komponentit ovat sinisellä.*

Kuvassa 10 esitetään HcOSA:n toiminnan kannalta tärkeimpien luokkien yhteyksiä. Luokat ovat väritetty sen mukaan mihinkä komponenttiin ne kuuluvat. Luokkien nimen alapuolella oleva <multiple> tarkoittaa, että HcOSA:ssa on useita vastaavia luokkia.

MainRestController ottaa vastaan REST serverille tulevan viestin ja ohjaa sen oikealle Endpoint-luokalle, esimerkiksi FormulaEndpoint toteuttaa kaavaan liittyvän viestin. Endpoint purkaa viestin ja lähettää sen Analyzer-luokalle. Tämä muuntaa EngineMsg-luokan avulla viestin HcEngine:lle sopivaksi ja antaa sen EngineInterface-luokalle. Tämä luokka hallitsee yhteyksiä HcEngine:lle ja välittää viestin EngineConnection-luokalle, joka sitten lähettää sen Socket-luokan avulla HcEngine:lle. Vastauksen tullessa EngineCollection valitsee viestin otsikkotietojen perusteella oikean MsgHandler-luokan, jolla

viesti luetaan, esimerkiksi FormulaMsgHandler-luokan. EngineInterface-luokka kerää vastaukset kaikilta EngineConnection-luokilta ja antaa ne Analyzer-luokalle. Lähtevistä ja vastaanotetuista viesteistä pidetään kirjaa EngineLog-luokan avulla. Mikäli viestejä on tullut usealta HcEngine:ltä yhdistetään ne Aggregator-luokan avulla. Viesti palautetaan esimerkiksi FormulaEndpoint-luokalle, joka muuntaa viestin HcGUI:lle sopivaan muotoon. InternalLog-luokan avulla pidetään lokia ohjelman yleisestä toiminnasta, esimerkiksi milloin HcEngine yhteys on luotu tai katkaistu.

Jos HcEngine lähettää hälytyksen, EngineConnection vastaanottaa sen, lukee sen AlarmMsgHandler:n avulla ja lähettää sen omalle ConnectionAlarms-luokalleen. Täällä hälytys tutkitaan, kirjataan muistiin ja lähetetään jonoon AlarmHandler-luokalle. Tämä luokka käsittelee kaikilta HcEngine:iltä tulleet hälytykset ja päättää mitä niille tehdään. Päätöksenteossa voidaan kysyä apua toisilta ohjelmilta ClientInterface-luokan kautta. Samoin päätöksestä johtuvassa toimenpiteessä voidaan hyödyntää ClientInterface-luokkaa, esimerkiksi sen kautta voidaan kutsua CollectionClientSender-luokkaa, jonka avulla voidaan ottaa yhteys Collector-ohjelmaan ja käskeä sitä aloittamaan tietojen keräys suoraan tukiasemasta.



Kuva 10. *HcOSA:n tärkeimpien luokkien yhteydet.*

6. TESTAUS

Ohjelman ja ohjelmiston testauksessa käytettiin monenlaisia erilaisia testejä, mutta niiden laajuus ja kattavuus vaihteli. Ohjelmistoa testattiin myös erilaisissa ympäristöissä

6.1 HcOSA:n testaus

Projektin alussa painopiste oli prototyypin nopeasti valmiiksi saamisessa ja testaus jäi taka-alalle. Yksikkötestaus on helpompi aloittaa heti projektin alussa, kun ei ole vielä paljon testattavaa. Näin ei kuitenkaan tässä tapauksessa käynyt ja mitä pidemmälle toteutuksessa päästiin, sitä enemmän oli yksikkötestejä tekemättä ja pienempi halu aloittaa niiden kirjoittamisurakka. Myös monet HcOSA:n luokista ja funktioista olivat erittäin riippuvainen toisista ohjelmista, joten yksikkötestien kirjoitus ei ollut yksinkertaista, sillä olisi pitänyt kirjoittaa myös tynkätoteutukset HcEngine:lle ja HcGUI:lle. Lopulta yksikkötestejä kirjoitettiin luokkien rakentajille ja purkajille sekä joillekin funktioille, joiden virhetilanteita oli muuten vaikea testata.

HcOSA:n integraatio- ja järjestelmätestaus suoritetaan omassa kehitysympäristössä. Ympäristössä on ajossa HcEngine, jolla on pieni testidata käytössään. HcGUI:lta tulevia komentoja simuloidaan lähettämällä REST-kutsuja Postman-ohjelmalla HcOSA:n REST rajapintaan. Postman on API-asiakas-ohjelma, jolla pystyy lähettämään viestejä toisten ohjelmien rajapintoihin ja näkemään niiden antamat vastaukset. Testaus aloitetaan käynnistämällä HcOSA ja tutkimalla onnistuiko se luomaan yhteyden HcEngine:n kanssa. Seuraavaksi lähetetään komentoja REST rajapintaan Postmanin avulla ja varmistetaan, että kaikki toimii oletetusti. Tämän prosessin automatisoinniksi Postmaniin pystyttiin rakentamaan testikokoelmia, jotka ajettaessa Postman vertaa HcOSA:lta saatuja vastauksia sille ennalta-annettuihin vastauksiin ja ilmoittaa niiden vastaavuudesta.

Mikäli virheen tapahtuessa sen syy ei ollut suoraan selvä, aloitettiin sen etsintä tutkimalla HcOSA:n pitämiä lokeja. Ohjelman ajaminen pysäytyspisteiden (breakpoint) kanssa toimi virheen tarkan paikallistamisen lopullisena keinona. Jos HcOSA:ssa tapahtunut virhe oli niin vakava, että se kaatoi koko ohjelman, voitiin se ajaa suoraan debug-tilassa, jolloin kaatumisen tapahtuessa nähtiin suoraan mistä se johtui.

Omassa kehitysympäristössä ohjelmat ajettiin suoraan binääritiedostoista. HcOSA, kuten muutkin ohjelmat, tuli pyöriä Docker-konteissa, joiden käyttöjärjestelmä ja kirjastoversiot erosivat kehitysympäristön vastaavista. Jotkin kirjastot saatiin päivitettyä konttiin, mutta joitakin muutoksia ei tuettu. Jotta HcOSA toimisi samanlailla testiympäristöissä ja

asiakkaan ympäristöissä kuin omassa ympäristössä, muutettiin kehitysympäristöä vastaamaan kontin ympäristöä.

Varsinaisen koodin laadun varmistamiseen ja testaamiseen käytettiin staattista koodin analysointia. Tämän testin avulla löydettiin esimerkiksi joitain alustamattomia muuttujia ja mahdollisuuksia muistivuodoille. Jotta koodi on aina tarkastettu, ajetaan staattinen analyysi joka yö koodille.

6.2 HealthCheck-ohjelmiston testaus

HealthCheck-ohjelmiston eri osat HcGUI, HcOSA ja HcEngine olivat kaikki eri henkilöiden kehittämiä ja omia projektejaan. Ohjelmien kehitys kuitenkin tapahtui saman aikaisesti ja hyvin lähekkäin. Täten ohjelmia oli helppo testata toisiinsa nähden ja kehitystyö olikin jossain suhteessa jatkuvaa järjestelmän integraatiotestausta. Testauksen helpottamiseksi ohjelmiin kehitettiin myös ainoastaan testaukseen tarkoitettuja ominaisuuksia. Esimerkiksi HcGUI:n kautta pystyttiin käskyttämään HcEngine:ä keräämään talteen sille tulevan puhelutiedon. Tämän tiedon sekä HcOSA:n ja HcEngine:n valmiiksi kirjoittamien lokien avulla pystyttiin paikallistamaan virheiden syyt. Myöhemmin HcOSA aktivoi tiedon talteen kirjoittamisen automaattisesti havaitessaan virheen.

Ohjelmiston testaus aloitettiin omassa testausympäristössä. Ympäristössä oli yksi reuna- ja yksi keskusserveri ja kaikki ohjelmat olivat Docker-konteissa. Tukiasemilta tulevaa liikennettä simuloitiin ohjelmilla, jotka lähettivät oikeista verkoista kerättyä dataa luupissa reunaserverillä sijaitsevalle vastaanotto-ohjelmalle, jolta HcEngine sai datansa. Testauksessa käytettiin perusjoukkoa kaavoista ja laskennoista, joiden avulla pystyttiin varmistamaan laskennan ja toiminnan kattavuus. Tätä joukkoa laajennettiin tarpeen tullen. GUI:n kautta pystyttiin suorittaan myös erilaisia testejä koko ohjelmistolle. Tässä ympäristössä varmistettiin prototyyppiversion päästä päähän toimivuus.

Seuraavassa vaiheessa HealthCheck:n testaus suoritettiin yrityksen omassa testilaboratoriossa, jossa oli oikeita tukiasemia. Liikenne näihin tukiasemiin tuli kuitenkin testipuheluista, joten data ei täysin vastaa oikeaa ympäristöä. Ohjelmistoa päästiin myös testaamaan asiakkaiden testilaboratorioissa ja joissakin näistä tukiasemien liikenne tuli oikeista puheluista. Vaikka omissa testilaboratorioissa data ei tullutkaan oikeista puheluista, vastasi sen toiminta oikeata ympäristöä niin hyvin, ettei asiakkaiden ympäristöistä ole löytynyt virheitä, joita ei olisi voinut toistaa omissa ympäristöissä.

Ohjelmiston testaamiseen käytetyt ympäristöt ovat sisältäneet hyvin pienen määrän (alta kymmenen) oikeita tukiasemia. Omissa testiympäristöissä, joissa liikenne generoidaan,

pystytään simuloimaan satoja tukiasemia. Kuitenkin varsinkin yksinkertaisessa simuloinnissa data on hyvin samanlaista ja tulee tasaisella nopeudella. Oikeissa ympäristöissä data ja sen määrä vaihtelee ajoittain merkittävästikin.

7. ARVIOINTI

Tässä luvussa pohditaan projektin ja sen eri vaiheiden onnistumista. Lisäksi käydään läpi, miten projektia tullaan jatkamaan.

7.1 Prosessin arviointi

Projektissa lähdettiin toteuttamaan täysin uutta ohjelmistoa. Esikuvana HealthCheck-ohjelmistolle toimivat vanhat vastaavaa toiminnallisuutta suorittaneet ohjelmistot. Lähtökohtaisesti HealthCheck-ohjelmiston tuli pystyä tekemään ajantasaista seurantaa ja analyysia matkapuhelinverkoille. Tämän mahdollistamiseksi tiedon käsittely toteutettiin striimin avulla sen sijaan, että se olisi tallennettu suuriin tietokantoihin kuten aiemmin. Tämän avulla erityisesti HcEngine pystyi suorittamaan laskut tehokkaasti ja kevyesti, mahdollistaen ajantasaisen laskennan. Vanhojen ohjelmistojen pohjalta voitiin toteuttaa samaa vanhaa perustoiminnallisuutta, mutta kehitetty nopea laskenta mahdollisti myös täysin uusien ominaisuuksien innovoinnin. Tämänkaltaisia ominaisuuksia olivat esimerkiksi ajantasaiset hälytykset ja näiden pohjalta tehty tukiasemien nopeasti katoavien lokien automaattinen keräys. Kehityksen pohjalta tehdyt innovaatiot tekivät kuitenkin ohjelmiston kehityksen vaikeasti ennakoitavaksi.

Projektissa hyödynnettiin tiimille ennen tuntematonta ketterää ohjelmistokehitysmallia. Yksittäisten iteraatioiden sijaan mallissa oli iteraatiojaksoja, jotka koostuivat useista iteraatioista. Suunnittelu ja työnjako tapahtui yksi jakso kerrallaan. Ohjelmiston kehitys oli alkanut vain hieman aiemmin ohjelmistokehitysmallin käyttöönottoa. Ensimmäiseen iteraatiojaksoon lähdettäessä ohjelmistosta ei kuitenkaan ollut vielä edes toimivaa prototyyppiversiota. Ensimmäisiin iteraatioihin oli laitettukin paljon tavoitteita ja työtehtäviä prototyypin valmiiksi saamiseksi. Evolutiivista prototypisointia hyödyntävälle projektille tyypilliseen tapaan kehitystyön edetessä opittiin koko ajan uutta ohjelmistosta sekä sen vaatimuksista. Opittujen asioiden pohjalta ohjelmistoon tai ohjelmaan jouduttiin tekemään muutoksia tai luomaan kokonaan uusia ominaisuuksia ja vaatimuksia. Jotkin näistä uusista ominaisuuksista tuli lisätä ennen kuin voitiin edetä seuraavissa iteraatioissa odottaviin työtehtäviin. Koska tätä tapahtui paljon ja iteraatiot olivat täyteen pakattuja, ei alkuperäisessä aikataulussa pysytty kiinni. Ymmärrettiin kuitenkin, että kehitystyön alkuvaiheessa olevalle projektille uusien ominaisuuksien innovointi ja kehitys oli tärkeämpää kuin pyrkimys noudattaa vanhentunutta suunnitelmaa.

Iteratiivinen kehitystyö sopii hyvin evolutiiviseen prototypisointiin, mutta paremman onnistumisen vuoksi suurta määrää iteraatioita ei tulisi suunnitella ennakkoon, vaan pohjata suunnitelmat edellisen tai muutaman edellisen iteraation tuloksiin. Varsinkin tämän tyyppisen projektin alussa muutoksia tulee erittäin paljon. Aikataulujen pitävyyden vuoksi tuli seuraavissa iteraatiojaksoissa pystyä ennakoimaan paremmin uusien ominaisuuksien nousemista kehitystyöstä. Jättämällä hyvin aikaa iteraatioihin mahdollisille uusille ominaisuuksille ja projektin alkuun nähden vähentynyt uusien ominaisuuksien keksimisen määrä auttoivat pitämään aikataulut paremmin kasassa. Silti tämänkaltaisessa projektissa on vaikea sanoa, mitä monen kuukauden päästä tulisi tehdä. Pienemmät uudet muutokset saatiin tehtyä iteraatiojakson aikana ja suuremmat voitiin ottaa huomioon seuraavaa jaksoa suunnitellessa. Projektin edetessä ja ohjelmiston kehittyessä myös ajanarviointikyky parantui jatkuvasti ja pian pystyttiin luottamaan siihen, että vaatimukset saatiin tehtyä annetussa ajassa.

7.2 Vaatimusten täyttyminen

Tärkeimpänä lähtökohtana ohjelmistolle oli kyky seurata ja analysoida matkapuhelinverkkojen terveydentilaa ajantasaisesti. Täydelliseen ajantasaisuuteen on mahdoton päästä, koska laskuissa menee aina oma aikansa. Koettiin myös, ettei ole järkeä laskea tuloksia täysin jatkuva-aikaisesti, sillä tämä olisi sekä erittäin raskasta että sen tuoma lisäarvo olisi kyseenalaista. Matkapuhelinverkoissa liikenne on usein epätasaista ja usein parempi kuva verkon tilasta saadaan laskettaessa arvoja pidemmältä aikaväliltä. Pienin tuettu laskenta-ajan tarkkuus ohjelmistossa on yksi minuutti. Tämä tarkoittaa, että laskennassa käytetään aina vähintään yhden minuutin sisällä saatua tietoa verkosta. Ohjelmiston itse laskentaan käyttämä aika riippui pitkälti suoritettavasta laskusta ja sen laajuudesta. Esimerkiksi ajantasaisessa seurannassa käytettyyn edellisen minuutin arvon laskentaan vastaus saadaan alle sekunnissa. Täten vaatimusta ajantasaisesta matkapuhelinverkon tilan seurannasta voidaan pitää onnistuneena.

Ohjelmiston toimintaympäristöksi vaadittiin Linux. Työn kirjoittajalle tämä ei ollut ongelma, koska hän oli tottunut kehittämään ohjelmia tässä ympäristössä. Kuitenkin HcEngine:n kehittäjä oli tottunut toimimaan Windows-ympäristössä ja käytti tässäkin ohjelmassa Windows 7 -käyttöjärjestelmää kehitysympäristönään. Ohjelma kyllä käännettiin Linux-ympäristöön sopivaksi, mutta joskus ohjelman virheellinen toiminta johtui käyttöjärjestelmien eroista. Docker ja kontit olivat kaikille uusia asia ja niiden opetteluun meni oma aikansa. Myös Dockeriin vaikuttava vaatimusmuutos, jonka takia käytettävissä olevat kirjastoversiot vaihtuivat, aiheutti lisätyötä. Ohjelmat saatiin kuitenkin toimimaan Linux-käyttöjärjestelmässä pyörivissä Docker-konteissa.

Vaatimus rajapintojen toteuttamisesta ja kyvystä keskustella HcGUI:n ja HcEngine:n kanssa oli oleellinen osa HcOSA:aa. Työn kirjoittajalla ei ollut aiempaa kokemusta laajojen rajapintojen toteutuksesta, joita toiset kehityksen alla olevat ohjelmat tulevat käyttämään. Tämän vuoksi tuli hieman yllätyksenä rajapintoihin tarvittavien muutosten määrä työn edetessä. Aluksi muutosten tekeminen oli työlästä, mutta sitä saatiin helpotettua jakamalla toteutusta paremmin hallittaviin osiin ja käyttämällä rajapintoja sekä periyttämistä.

HcOSA:n kyky toimia useamman HcEngine:n kanssa jäi vajaaksi. Prototyypin jälkeen HcOSA laajennettiin toimimaan usean laskentayksikön kanssa. Se pystyi muodostamaan monta yhteyttä, luomaan kaavoja ja laskentoja rinnakkaisesti eri HcEngine:ille. Se pystyi myös yhdistämään sen aikaisten laskentojen tulokset yhdeksi koko verkkoa vastaavaksi tulokseksi. Laskennan monimutkaistuttua HcOSA ei enää tukenut tulosten yhdistämistä ja siten vaatimusta ei voida pitää täysin toteutuneena. Kuitenkin ensimmäinen julkaisuversio on tarkoitettu pienille verkoille, jolloin tämä vaatimus ei ole tarpeellinen.

HcOSA:n kyky vian havaitessaan aktivoida automaattisesti tarkka keräys tukiasemasta herätti mielenkiintoa ja siitä tulikin ohjelman pääominaisuus ensimmäiseen julkaisuversioon. HcOSA:aan ja HealthCheck-ohjelmistoon ollaan erittäin tyytyväisiä sekä sisäisesti että asiakkaankin päässä. Myös yrityksen toinen hälytyksiin liittyvä ohjelma käyttää HealthCheck:in luomia hälytyksiä HcOSA:n REST API:n kautta.

7.3 Suunnittelun arviointi

Prototyypin nopeasti valmiiksi saamiseksi projektin alussa ei suunnitteluun käytetty paljon aikaa. Työn tekijällä ei ollut myöskään laajaa kokemusta vastaavasta projektista. Suunnittelun vähyys, toteutuksen priorisointi ja kokemuksen puutteesta johtunut ennakkokyvyttömyys johtivat HcOSA:n prototyyppiversion kyseenalaiseen toteutukseen. Suunnittelussa oli keskitytty helppoon laajennettavuuteen. Uutta toteutusta olikin helppo lisätä, mutta ajan säästämiseksi se usein lisättiin olemassa olevaan luokkaan. Pian jotkin luokista olivat paisuneet ja vastasivat liian monesta asiasta. Lisäksi ohjelman rakenteessa oli paljon riippuvuuksia. Näiden vuoksi yhteen luokkaan tehdyt muutokset vaikuttivat moniin muihin luokkiin ja täten muutosten tekeminen oli työlästä. Jatkokehityksen kannalta oli tärkeää korjata toteutus. Korjaamisessa käytettiin apuna SOLID-suunnitteluperiaatteita.

Yhden vastuualueen periaatteen (single responsipility principle) pohjalta suurten luokkien eri vastuualueet jaettiin omiksi luokikseen muutosten helpottamiseksi. Luokkien riip-

puvuutta tietystä toteutuksesta vähennettiin riippuvuuksien syötön (dependency injection) avulla. Nyt muutoksia tehtäessä ei jouduttu koskemaan niin moneen paikkaan. Silti muutettaessa REST tai HcEngine-rajapintaa saattoivat muutokset vaikuttaa useampaankin paikkaan. Kuitenkin koettiin, että tämä tapahtuisi riippumatta siitä minkälaista suunnittelua käytettiin tai suunnitteluun käytettävä aika olisi niin suuri, ettei se tässä vaiheessa projektia olisi enää järkevää.

Vaikka ohjelma ei täysin noudattanut avoin/suljettu periaatetta (open/closed principle), niin silti periaatteen hengen mukaisesti uuden toiminnallisuuden lisääminen ei vaikuta jo olemassa olevaan toiminnallisuuteen. Esimerkiksi uuden päätepisteen lisääminen REST API:in vaatii sen toteuttavan Controller-luokan lisäyksen ja MainRESTController-luokalle täytyy kertoa, että lisätty luokka hoitaa tähän päätepisteeseen tulevat viestit. Mikäli tämä uusi endpoint vaatii uudentyyppistä tietoa, tulee HcEngine-rajapintaa lisätä tälle tarvittavat käsittelijät. Olemassa oleviin toteutuksiin saatettaan joutua koskemaan ainoastaan siinä tapauksessa, jos HcEngine tai HcGUI muuttavat rajapintatoteutuksiaan uuden toiminnallisuuden takia.

7.4 REST API:n arviointi

REST on ohjelmistoarkkitehtuurimalli web-rajapintojen toteuttamiseen. Se esittää kuusi vaatimusta rajapinnan toteuttamiselle, ja API:a, joka noudattaa näitä kaikkia, voidaan kutsua RESTful tai REST API:ksi. Monet REST rajapinnoista eivät kuitenkaan täytä näitä kaikkia vaatimuksia, mutta silti usein niitä kutsutaan nimellä RESTful API tai REST API. [1,11]

HcOSA noudattaa kahta ensimmäistä vaatimusta clientin ja serverin erosta sekä serverin tilattomuudesta. HcGUI toteuttaa clientin ja HcOSA serverin sekä molempia voidaan kehittää itsenäisesti. Täten client ja server ovat erossa. Tilattomuusvaatimuksen HcOSA täyttää olemalla tallentamatta asiakkaan tilaa muistiin. Kaikki HcOSA:lle tulevat kutsut käsitellään edellisistä kutsuista riippumattomina. HcOSA ei noudata vaatimusta välimuistin (cache) käytöstä, koska REST API on pystynyt palvelemaan helposti tähän asti tulleen liikenteen ja varsinaista tarvetta välimuistin tukemiselle ei ole ollut. API:n kautta pyydettävä tieto vanhenee jatkuvasti, eikä sitä siten ole järkevää tallentaa välimuistiin. Koska API ei tue välimuistia, ei sen tarvitse huolehtia välimuistin ylläpidosta. HcOSA noudattaa osittain vaatimusta yhtenäisestä rajapinnasta. Eri resurssit ovat haettavissa uniikin osoitteen ja HTTP-metodin avulla, mutta paluuviesteissä ei kuitenkaan kerrota toisista käytettävissä olevista päätepisteistä. Tämä johtuu siitä, että HcGUI kehitettiin samaan aikaan ja tieto eri päätepisteistä oli helpompi välittää dokumentin avulla.

HcOSA:ssa noudatetaan kerrosomaisuutta (layered), sillä REST API on tekimisissä ainoastaan yhden HcOSA:n kerroksen kanssa eikä sille kerrota mitä HcOSA:n tuli tehdä vastauksen saamiseksi. Viimeisenä on vapaaehtoinen vaatimus, jonka mukaan serverin pitäisi pystyä lähettämään suoritettavissa olevaa koodia pyydettäessä. Tätä ei tueta, koska sille ei ole löytynyt käyttötarkoitusta. [11]

HcOSA:n REST API ei siis täytä kaikkia REST arkkitehtuurin vaatimuksia. Huolimatta näistä puutteista HcOSA:n rajapinta toimii ohjelmiston tarpeiden mukaisesti. Jatkon kannalta HcOSA:n REST API:a voitaisiin laajentaa palauttamaan linkkinä päätepisteelle oleelliset toiset päätepisteet. Tämän avulla voitaisiin kertoa asiakkaalle dynaamisesti kaikki käytettävissä olevat päätepisteet, kunhan asiakas tietää edes yhden päätepisteistä. Tällöin asiakkaan ei tarvitse tuntea kaikkia päätepisteitä. Jatkossa REST API voisi tukea myös joidenkin vähemmän muuttuvien resurssien tallentamista välimuistiin. Välimuisti tulisi tässä tapauksessa tyhjentää aina muutosten tullessa, jotta asiakas ei saisi vanhentunutta tietoa.

7.5 Testauksen arviointi

Kiireen vuoksi yksikkötestien kirjoittamista ei aloitettu heti projektin alussa ja myöhemmin testien kirjoittamiseen vaadittu työmäärä oli kasvanut niin suureksi, ettei sitä haluttu aloittaa. Testien kirjoittamisesta olisi ollut kyllä paljon apua. Niiden avulla olisi ollut paljon helpompaa paikallistaa virheet tarkasti kuin käytetyt integraatio- ja järjestelmätestit. Myös regressiotestauksessa niiden avulla olisi saatu heti tietää, jos uusi toiminnallisuus rikkoi jo olemassa olevan toiminnallisuuden.

Molemmissa sekä simuloiduissa että oikeissa testiympäristöissä on testattu ensimmäistä julkaisuversiota varten yhden reuna- ja keskusserverin välistä toimintaa vain rajatulla määrällä tukiasemia. Ensimmäisen HealthCheck-ohjelmiston julkaisuversion on tarkoitus palvella noin sataa tukiasemaa. Julkaisua varten erillinen testaustiimi suoritti turvallisuus- ja suorituskykytestit omissa simuloiduissa ympäristöissään. HealthCheck-ohjelmisto läpäisi nämä testit ja toimi jopa 1000 tukiasemalla, mikä on yhden reunaserverin tukema maksimimäärä.

7.6 Jatkokehitysideat

HealthCheck-ohjelmiston kehitys on jatkunut ensimmäisen julkaisuversion jälkeen. Seuraavissa versioissa ohjelmistoon on tarkoitus tuoda laajojen matkapuhelinverkkojen tukeminen, koneoppimisanalyysi sekä tukiasemasta saatavien uusien raporttityyppien tukeminen.

Monien HcEngine:iden tukeminen HcOSA:ssa vaatii tulosten yhdistämisen uudelleentoteuttamisen. HcOSA:n tulee laskea koko matkapuhelinverkkoa vastaavat arvot, mutta tätä varten HcEngine:n tulee palauttaa sen laskennassaan käyttämä kaava ja tarvittavat arvot. HcOSA tulee osata tulkita tämä kaava, sijoittaa siihen kaikilta HcEngine:iltä saadut arvot ja suorittaa laskutoimitus. HcGUI:ssa tarvitsee myös ottaa huomioon tulosten tulemien useammasta HcEngine:stä, esimerkiksi tulee lisätä mahdollisuus valita mitä HcEngine:ä halutaan ohjata.

Ohjelmistossa olisi mahdollisuus hyödyntää koneoppimista. Koneoppimisanalyysin hoitaisi toinen ohjelma ja HcOSA voisi kysyä tältä apua, esimerkiksi päätellessään, mitä hälytykselle tulisi tehdä. Mahdollista olisi myös, että koneoppimisohjelma voisi tehdä hälytyksen HcOSA:lle huomatessaan poikkeaman ja HcOSA tekisi oman analyysinsa sille.

Uudet raporttityypit vaativat lisäyksiä HcEngine:en, jonka tulee osata purkaa raporttityypin mukainen viesti ja tehdä tälle laskenta. HcGUI:hin tulisi lisätä kyky näihin uusiin raportteihin liittyvien laskentojen luomiseen ja näkymät uusille tuloksille. HcOSA:lle tämä ei välttämättä aiheuta lisätyötä vastausten dynaamisen käsittelyn vuoksi. Jos uusi data sisältää jotain todella poikkeavaa tai sille tehtävät laskennat ovat huomattavasti erilaisia, joudutaan HcOSA:kin päivittämään.

Uusien ominaisuuksien testitapauksien tekemisen lisäksi testausta tulisi kehittää parantamalla yksikkötestien kattavuutta ja luomalla testiautomaatiota, jonka avulla voitaisiin varmistaa jokaisen version toimivuus. Monen HcEngine:n testaaminen vaatii myös olemassa olevien testausympäristöjen laajentamista tai kokonaan uuden sellaisen. Ympäristöön tarvitaan useita reunaservereitä, jotta HcOSA:n toiminta voidaan varmistaa.

Mikäli HcOSA olisi pullonkaula HealthCheck-ohjelmistossa tulisi sen pystyä skaalautumaan paremmin. Tämä voidaan saada aikaiseksi käynnistämällä useampi HcOSA-ohjelma ja toteuttamalla kuormaintasain (load balancer) HcOSA:n ja GUI:n välille. HcEngine puoltaa tämänkaltaista skaalautuvuutta tukemalla useampaa HcOSA-yhteyttä. Samoin HcOSA tukee suurilta osin tämänkaltaista skaalautuvuutta. Ainoastaan hälytysten käsittelyä ei tue tätä, sillä se tarvitsee päätöksenteossaan muistissaan pitämää hälytyshistoriaa. Tämä historia tulisi tallentaa tietokantaa, johon kaikilla HcOSA-ohjelmat pääsevät kiinni. Muuten HcOSA:n toimivuuden kannalta ei ole väliä, kunhan kuormantasain huolehtii siitä, että HcOSA:n sisäisten parametrien muutokset lähetetään kaikille ohjelmille.

Vaikka HcGUI ja HcEngine tukisivat kaksinkertaista nopeutta mihin yhdellä HcOSA:lla päästään, ei toisen HcOSA:n käynnistys kuitenkaan tarkoita HealthCheck-ohjelmiston nopeuden kasvamista kaksinkertaiseksi. Skaalautuvuus olisi riippuvainen siitä, kuinka

hyvin hälytyksen käsittely saadaan rinnastetusti. Suurin pullonkaula tässä todennäköisesti on lisätietojen pyytäminen muista ohjelmista.

8. YHTEENVETO

Tässä työssä esiteltiin HealthCheck-ohjelmisto, joka pystyy laskemaan ja analysoimaan matkapuhelinverkosta tulevaa dataa reaaliajassa sekä tarjoamaan graafisen käyttöliittymän tulosten tutkimiseen ja ohjelman hallintaan. Työssä myös toteutettiin ohjelmistoon kuuluva ohjelma, joka vastasi tiedon yhdistämisestä ja analysoinnista. Ohjelma vastasi myös viestien välittämisestä HealthCheck-ohjelmiston eri osille sekä toisille ohjelmistoille, joiden avulla muun muassa saatiin tehtyä tarkennettuja tiedon keräyksiä suoraan tukiasemista.

Alussa kiireellä tehty prototyyppimainen ohjelman ensimmäinen versio ei antanut paljon aikaa suunnittelulle taikka testaukselle. Tämä johti pieniin vaikeuksiin projektin edetessä, mutta suunnittelumuutoksilla saatiin ohjelma helpommin muunnettavaksi. Myös vaatimusten muuttuminen ja lisävaatimukset toivat oman lisänsä ohjelman suunnitteluun ja toteutukseen. REST API suunniteltiin ja toteutettiin HcGUI:n tarpeita varten. Tämän vuoksi jotkin REST arkkitehtuurin vaatimuksista jäi täyttymättä, kun niille ei ollut suoranaista tarvetta HcGUI:n kanssa kommunikoinnissa. Testauksessa yksikkötestit jätettiin taka-alalle ja keskityttiin enemmän ohjelman kokonaisuuden testaukseen osana HealthCheck-ohjelmistoa. Vaikeuksista huolimatta ohjelma ja ohjelmisto täyttivät julkaisuun asetetut tavoitteet ja sekä tiimi ja asiakkaat ovat tyytyväisiä ohjelmistoon.

LÄHTEET

[1] A. Avram, Why Some Web APIs Are Not RESTful and What Can Be Done About It, InfoQ, 2014. Saatavissa: https://www.infoq.com/articles/web-api-rest/

[2] GSMA Intelligence, The Mobile Economy 2019, GSM Association, 2019. Saatavissa: https://www.gsmaintelligence.com/research/?file=b9a6e6202ee1d5f787cfebb95d3639c5&download

[3] O. Ibe, Fundamentals of Data Communication Networks, Wiley, 2017, Chapter 11 – Introduction to Mobile Communications Networks

[4] D. Kreutz, F. M. V. Ramos, P. E. Veríssimo, C. E. Rothenberg, S. Azodolmolky and S. Uhlig, "Software-Defined Networking: A Comprehensive Survey," in Proceedings of the IEEE, vol. 103, no. 1, pp. 14-76, Jan. 2015.

[5] Krill, P, Code it again, Sam: Microsoft's Casablanca ties C++ to the cloud, InfoWorld.Com, 2012. Saatavissa: https://www.infoworld.com/article/2616939/code-it-again--sam--microsoft-s-casablanca-ties-c---to-the-cloud.html

[6] I. Mejía, Modern C++ micro-service implementation + REST API, Medium, 2017. Saatavissa: https://medium.com/audelabs/modern-c-micro-service-implementation-rest-api-b499ffeaf898

[7] I. Mejía, Modern C++ micro-service implementation + REST API Part II, Medium, 2017. Saatavissa: https://medium.com/audelabs/modern-c-micro-serivce-rest-api-part-ii-7be067440ca8

[8] Microsoft, C++ REST SDK GitHub. Saatavissa: https://github.com/microsoft/cpprestsdk

[9] Microsoft, Parallel Patterns Library (PPL), 2016. Saatavissa: https://docs.microsoft.com/en-us/cpp/parallel/concrt/parallel-patterns-library-ppl?view=vs-2019

[10] D. Murray, T. Koziniec, K. Lee and M. Dixon, "Large MTUs and internet performance," *2012 IEEE 13th International Conference on High Performance Switching and Routing*, Belgrade, 2012, pp. 82-87.

[11] P. Raj and H. Subramanian, *Hands-on RESTful API Design Patterns and Best Practices.* 2019

[12] M. Stefani, Pistache. Saatavissa: http://pistache.io/

[13] Säteilyturvakeskus, Matkapuhelinverkon toiminta ja tukiasemat. Saatavissa: https://www.stuk.fi/aiheet/matkapuhelimet-ja-tukiasemat/matkapuhelinverkko/matkapuhelinverkon-toiminta-ja-tukiasemat

[14] P. Thoman, P. Gschwandtner and T. Fahringer, "On the Quality of Implementation of the C++11 Thread Support Library," *2015 23rd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, Turku, 2015, pp. 94-98.

LIITE A: HCOSA REST API

Alarms

Alarms-resurssilla käsitellään hälytyksiä.

GET /alarms

Haetaan kaikki hälytykset. Palautetaan 200 OK ja lista hälytyksistä tai 404 Not Found.

GET /alarms?source={source}&status={status}&start={start}&end={end}

Haetaan hälytykset, jotka vastaavat kyselyssä annettuja suodattimia. Source kertoo hälytyksen lähteen (esim. hcengine), status kertoo hälytyksen tilan (esim. active), start kertoo hälytyksen ajan alarajan epoch aikana ja end ylärajan. Palautetaan 200 OK ja lista hälytyksistä tai 404 Not Found.

DELETE /alarms

Poistetaan bodyssa annettu hälytys. Palautetaan 200 OK tai 404 Not Found.

Btsinfo

Btsinfo-resurssilla käsitellään tukiasemien tietoja.

GET /btsinfo

Haetaan tukiasemien tunnisteet, IP-osoitteet ja ohjelmistoversiot. Tarvitaan verkkotopologian esittämiseen. Palautetaan 200 OK ja lista tukiasemien tiedoista tai 404 Not Found.

Computations

Computations-resurssilla käsitellään laskentoja.

GET /computations

Haetaan kaikki HcEngine:ssä olevat laskennat. Palautetaan 200 OK ja lista laskennoista tai 404 Not Found.

POST /computations

Luodaan bodyssa olevien tietojen perusteella uusi laskenta HcEngine:en. Palautetaan 200 OK tai 400 Bad Request ja syy virheeseen (todennäköisesti laskenta on olemassa jo tai virheelliset arvot).

DELETE /computations

Poistetaan bodyssa annettu laskenta HcEngine:stä. Palautetaan 200 OK tai 404 Not Found.

Datatypes

Datatypes-resurssilla käsitellään HcEngine:n käytössä olevia raportti- ja tietotyyppejä

GET /datatypes

Haetaan tiedot kaikista HcEngine:n käytettävissä olevista raportti- ja tietotyypeistä. Näitä tietoja tarvitaan HcGUI:ssa uusien kaavojen muodostamisessa. Palautetaan 200 OK ja lista tai 404 Not Found.

Engineconfigs

Engineconfigs-resurssilla käsitellään HcEngine:n konfiguraatioparametrejä.

GET /engineconfigs

Haetaan kaikki HcEngine:n parametrit. Palautetaan 200 OK tai 404 Not Found.

PUT /engineconfigs

Päivitetään bodyssa annettu parametri annettuun uuteen arvoon. Voidaan ohjata HcEngine:n toimintaa. Palautetaan 200 OK tai 400 Bad Request ja syy (todennäköisesti huono arvo parametrille).

Formulas

Formulas-resurssilla käsitellään kaavoja.

GET /formulas

Haetaan kaikki HcEngine:n käytössä olevat kaavat. Palautetaan 200 OK ja lista tai 404 Not Found.

GET /formulas/groups

Haetaan kaikki HcEngine:n käytössä olevat kaavakokoelmat (monesta kaavasta muodostuva ryhmä, esimerkiksi kaikki Handovereihin liittyvät yksittäiset kaavat voidaan sitoa yhdeksi kaavakokoelmaksi, jolloin on helpompi tehdä laskentaa näille). Palautetaan 200 OK ja lista tai 404 Not Found.

POST /formulas

Luodaan uusi kaava bodyssa annettujen tietojen perusteella. Palautetaan 200 OK tai 404 Bad Request ja syy (todennäköisesti samanniminen kaava on jo olemassa).

POST /formulas/groups

Luodaan uusi kaavakokoelma bodyssa annetuista kaavoista. Palautetaan 200 OK tai 404 Bad Request ja syy (todennäköisesti samanniminen kaavakokoelma on jo olemassa).

DELETE /formulas

Poistetaan bodyssa annettu kaava. Palautetaan 200 OK tai 404 Not Found.

DELETE /formulas/groups

Poistetaan bodyssa annettu kaavakokoelma. Palautetaan 200 OK tai 404 Not Found.

Groups

Groups-resurssilla käsitellään ryhmiä, joille laskenta voidaan suorittaa.

GET /groups

Haetaan kaikki HcEngine:n käytettävissä olevat ryhmät. Palautetaan 200 OK ja lista tai 404 Not Found.

POST /groups

Luodaan uusi ryhmä bodyssa annettujen tietojen perusteella. Palautetaan 200 OK tai 400 Bad Request ja syy (todennäköisesti samanniminen ryhmä on jo olemassa).

DELETE /groups

Poistetaan bodyssa annettu ryhmä. Palautetaan 200 OK tai 404 Not Found.

Osaconfigs

Osaconfigs-resurssilla käsitellään HcOSA:n konfiguraatioparametrejä.

GET /osaconfigs

Haetaan kaikki HcOSA:n parametrit. Palautetaan 200 OK tai 404 Not Found.

PUT /osaconfigs

Päivitetään bodyssä annettu parametri annettuun uuteen arvoon. Voidaan ohjata HcOSA:n toimintaa. Palautetaan 200 OK tai 404 Bad Request ja syy (todennäköisesti huono arvo parametrille).

Results

Results-resurssilla käsitellään laskentatuloksia.

POST /results

Haetaan vastaus bodyssa annettuun laskentakyselyyn. Palautetaan 200 OK ja tulos tai 404 Not Found.

Sequences

Sequences-resurssilla käsitellään tilakoneita. Tilasiirtyminä toimivat laskennoille annettujen ehtojen täyttyminen. Näitä käytetään hälytysten luomisessa.

GET /sequences

Haetaan kaikki HcEngine:n käytettävissä olevat tilakoneet. Palautetaan 200 OK ja lista tai 404 Not Found.

POST /sequences

Luodaan uusi tilakone bodyssa annettujen ehtojen mukaan. Palautetaan 200 OK tai 400 Bad Request ja syy (todennäköisesti samanniminen tilakone on jo olemassa).

DELETE /sequences

Poistetaan bodyssa annettu tilakone. Palautetaan 200 OK tai 404 Not Found.

Statistics

Statistics-resurssissa käsitellään statistiikkaa.

GET /statistics

Haetaan HcOSA:n ja HcEngine:n toimintaan liittyvää statistiikkaa, esimerkiksi käynnissä oloaika, muistinkulutus ja vastaanotettujen viestien määrä. Palautetaan 200 OK ja statistiikka tai 404 Not Found.

Supervisors

Supervisors-resurssissa käsitellään valvojia, joidenka perusteella HcEngine luo hälytyksiä. Valvoja liittyy aina johonkin laskentaan tai tilakoneeseen. Valvojan ehtojen täytyttyä luodaan hälytys.

GET /supervisors

Haetaan kaikki HcEngine:ssä käynnissä olevat valvojat. Palautetaan 200 OK ja lista tai 404 Not Found.

POST /supervisor

Luodaan uusi valvoja bodyssa annettujen tietojen mukaan. Palautetaan 200 OK tai 400 Bad Request ja syy (todennäköisesti samanniminen valvoja on jo olemassa).

DELETE /supervisor

Poistetaan bodyssa annettu valvoja. Palautetaan 200 OK tai 404 Not Found.